CIC - IPN

Instituto Politécnico Nacional

Centro de Investigación en Computación

Secretaría de Investigación y Posgrado

GENARO:

Diseño y construcción de una plataforma para implementar una Línea de Productos de Software

T E S I S

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A

EL ING. RUBÉN MURGA TAPIA

DIRECTORES DE TESIS:

DR. ROLANDO MENCHACA MÉNDEZ

DR. LUIS EDUARDO LEYVA DEL FOYO

MÉXICO, D.F. **DICIEMBRE 2013**

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 12:00 horas del día 05 del mes de junio de 2013 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Diseño y construcción de una plataforma para implementar una línea de productos de software"

Presentada por el alumno:

Murga	**Tapia**	**Rubén**
Apellido paterno	Apellido materno	Nombre(s)

Con registro: B 9 9 1 2 6 2

aspirante de: **MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA
Directores de Tesis

Dr. Rolando Menchaca Méndez

Dr. Luis Eduardo Leyva del Foyo

Dr. Marco Antonio Moreno Ibarra

Dr. Rolando Quintero Téllez

Dr. Miguel Jesús Torres Ruiz

M. en C. Sandra Dinora Orantes Jiménez

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION EN COMPUTACION
DIRECCION

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la ciudad de México el día 05 del mes de Junio del año 2013, el(la) que suscribe Rubén Murga Tapia alumno (a) del Programa de Maestría en Ciencias de la Computación con número de registro B9912, adscrito al Centro de Investigación en Computación, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección del Dr. Rolando Menchaca Méndez y del Dr. Luis Eduardo Leyva del Foyo y cede los derechos del trabajo intitulado "Diseño y construcción de una plataforma para implementar una línea de productos de software", al Instituto Politécnico Nacional para su difusión, con fines académicos y de Investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección ruben.murgat@gmail.com / rolando.menchaca@gmail.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Rubén Murga Tapia

RESUMEN

La *línea de productos de software* (LPS) es un paradigma de ingeniería de software que integra áreas como la ingeniería de requerimientos, la ingeniería de dominio y la ingeniería de aplicación para intentar resolver el problema de la construcción masiva y eficiente de software. Su propósito es emular las líneas de producción industriales que son capaces de generar, de manera eficiente, grandes volúmenes de productos de alta calidad.

Dicho paradigma se basa en el principio de *familia de producto* en el cuál se hace énfasis en el reúso de componentes de software para resolver problemas similares, así como en el concepto de la *programación generativa*, que es un estilo de programación basado en la creación automatizada de código fuente por medio de plantillas genéricas, clases, prototipos, componentes y lenguajes de especificación de alto nivel.

En esta tesis se presenta el diseño, construcción y validación experimental de una herramienta para el desarrollo semi-automatizado de sistemas de información que está basada en los principios de la línea de productos de software. Bajo el esquema propuesto, una línea de producción de software para una familia de productos se instancía por medio de la definición de las componentes estáticas, que son comunes a todos los productos de la familia, y de las componentes dinámicas que varían entre los diferentes miembros de la familia. La herramienta propuesta está compuesta por un ambiente gráfico que asiste en la administración de diferentes componentes de software; un lenguaje para especificación de la parte variable, así como su respectivo intérprete y; un conjunto de plantillas que definen la parte estática de diferentes sistemas.

La herramienta propuesta se ha probado de forma extensiva, y un número importante de desarrollos obtenidos mediante ella están actualmente en operación tanto en el sector público como en el privado.

Abstract

A Software Product Line (SPL) is a paradigm that incorporates different areas of software engineering such as requirement engineering, domain engineering and application engineering to solve the problem of the massive and efficient construction of software products. Its main purpose is to emulate the way in which industrial product lines are capable of producing large volumes of high quality products.

This paradigm is based on the product family principle which emphasizes the application of software component reuse to solve similar problems, and the concept of generative programming to automatically generate high quality source code.

In this thesis we present the design, construction and experimental validation of a software tool that supports the process of automatized construction of information systems. The tool is based on the principles of the Software Product Lines. Under the proposed scheme, a software product line is instantiated by means of the definition of the static components of the system which are those logical parts that are common to all the members of the product family, and by the dynamic components that vary from one member to another. The proposed tool is composed by a management graphical user interface; a language, with its corresponding interpreter, for the specification of the variable components of the products; and a set of templates that define the static components of a series of information systems.

The proposed tool has been extensively tested and it has been used to develop an important number of systems that are currently under operation in both the private industry and the government.

AGRADECIMIENTOS

Agradezco al Centro de Investigación en Computación del IPN por el apoyo recibido durante mi estancia en sus instalaciones.

La presente Tesis es un esfuerzo en el cual, directa o indirectamente, participaron varias personas leyendo, opinando, corrigiendo, teniéndome paciencia, dando ánimo.

Agradezco a los Doctores Rolando Menchaca Méndez y Luís Eduardo del Foyo por haber confiado en mi persona, por la paciencia y por la dirección de este trabajo. Al Dr. Marco Antonio Moreno Ibarra y la Dra. Alma Delia Cuevas Rasgado por los consejos, el apoyo y el ánimo que me brindaron.

A mi esposa Eva y a mis hijos que me acompañaron en esta aventura que significó la maestría y que, de forma incondicional, entendieron mis ausencias y mis malos momentos. A mi padre, que a pesar de la distancia siempre estuvo atento para saber cómo iba mi proceso.

Gracias a todos.

Contenido

LISTADO DE FIGURAS

CAPÍTULO I

INTRODUCCIÓN

1. CONTEXTO

En este trabajo de tesis se aborda el paradigma de *Línea de Productos de Software*, cuyas bases son los procedimientos de reutilización del software. Como se muestra en la Figura 1, el concepto comienza en 1960 con la práctica de estructurar la solución de problemas en subrutinas que podían ser utilizadas en la resolución de otros problemas similares.

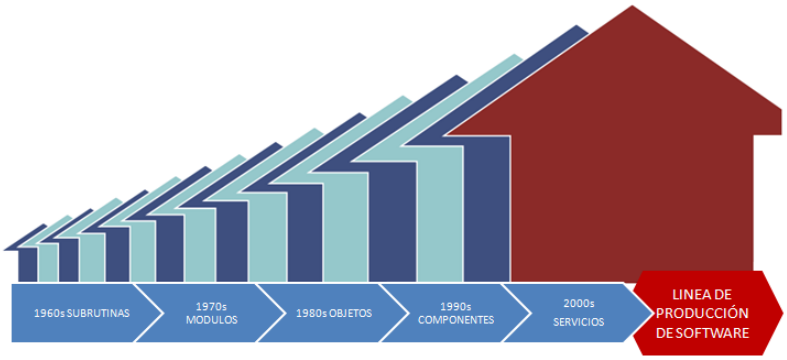

Figura 1. Historia del Reúso.

En el paradigma de Línea de Productos de Software, se encuentran inmersas técnicas y procedimientos de la Ingeniería de Requerimientos, Ingeniería de Dominio e Ingeniería de Aplicación. La Línea de Productos de Software (*LPS*) incorpora los conceptos centrales de la ingeniería de línea de producto tradicional, como son el uso de una plataforma y la habilidad de proveer una personalización en masa.

La definición más comúnmente aceptada de una *LPS* procede de [CLEMENTS NORTHROP 2001] y dice lo siguiente. "Se definen las Líneas del Producto de Software como un conjunto de sistemas de software, que comparten un conjunto común de características (*features*), las cuales satisfacen las necesidades específicas de un dominio o segmento particular de mercado, y que se desarrollan a partir de un sistema común de activos base (*core assets*) de una manera preestablecida".

El paradigma *LPS* se refiere a la producción de productos de una misma familia, y el éxito del desarrollo de una línea de producción depende de saber acotar las partes comunes y variables de un producto. Los términos de "Línea de producción de Software" y "Familia de Producto de Software", sólo se tratan de sinónimos utilizados en Norte América y Europa respectivamente [KLAUS GUNTER FRANK 2005].

De acuerdo con [NORTHROP 2002], el Instituto de Ingeniería de Software (SEI, por sus siglas en inglés) no sólo ha confirmado los beneficios de seguir el enfoque de Línea de

Productos de Software, sino que también encontró que el hacerlo es a la vez una decisión técnica y de negocios. Para tener éxito con las líneas de productos de software, una organización debe alterar sus prácticas técnicas, prácticas de gestión, la estructura organizacional del personal y el enfoque de negocios.

Respecto a los beneficios, en [SEI 2013] encontramos que los beneficios ayudan a las organizaciones a superar los problemas causados por la escasez de recursos. Organizaciones de todo tipo y tamaño han descubierto que cuando se aplica con habilidad una estrategia de línea de producto, puede producir muchos beneficios, y finalmente, dar a las organizaciones una ventaja competitiva. Ejemplo de beneficios organizacionales incluyen los siguientes.

- Aumento de la productividad.
- Aumento de la calidad de los productos.
- Disminución de costos.
- Disminución de las necesidades de mano de obra.
- Disminución del tiempo de comercialización.
- Aumento en la posibilidad de entrar en nuevos mercados.

Para [OSCAR SALVA 2010], las LPS pueden incrementar significativamente la productividad de los ingenieros de software, entendida como una reducción en el esfuerzo y el costo necesario para desarrollar, poner en marcha y mantener un conjunto de productos de software similares. En los casos de estudio se han observado mejoras en la productividad que duplican o triplican los enfoques artesanales. Los beneficios que las *LPS* aportan a la calidad se pueden medir de dos formas. La primera mediante el grado de precisión con que cada producto se ajusta a las necesidades de cada cliente y segundo, la tasa de defectos en los productos de la *LPS*.

Con respecto a las características principales de las líneas de producción de software, [KLAUS GUNTER FRANK 2005] enfatiza en la necesidad de una plataforma tecnológica para la personalización en masa de los productos. El enfoque de plataforma, permite a las manufacturas ofrecer una gran variedad de productos y al mismo tiempo reducir los costos.

La plataforma tecnológica no es todo lo que necesitamos en una *LPS*, también es importante el desarrollo de los activos principales y las actividades relacionadas con la administración. El diseño y construcción de la plataforma de *LPS* no sólo es una herramienta de generación de software tipo CASE, en lugar de esto, debe diseñarse desde un principio, considerando la administración de sus activos, la variabilidad en los productos y la gestión de varias líneas de producción.

Los activos principales forman la base de la línea de producción de software. Los activos principales incluyen a la arquitectura, los componentes de software reutilizables, modelo de dominios, instrucciones de requerimientos, documentación y especificaciones, modelos de desempeño, horarios, presupuestos, planes de prueba, casos de prueba, planes de trabajo y descripción de procesos [NORTHROP 2002].

Con el objetivo de generalizar, hay tres actividades esenciales y altamente relacionadas que mezclan la tecnología y las prácticas de negocio. Como se muestra en la Figura 2, el desglose de una línea de productos incluye el desarrollo de activos principales y el desarrollo de nuevos productos a partir de los principales activos en el marco de la gestión técnica y de organización. En la figura, las flechas de rotación indican que las actividades no sólo se limitan al desarrollo de productos, sino también a la revisión de los activos existentes o incluso activos nuevos.

Figura 2. Actividades esenciales de línea de producto.

Una comparación entre la eficiencia obtenida mediante el desarrollo artesanal y la línea de producción de software, lo podemos encontrar en [OSCAR SALVA 2010]. En la Figura 3 se ilustra la evolución en los costos cuando se utiliza desarrollo artesanal y cuando se utiliza una *LPS*. A mayor número de productos, mayor esfuerzo y costo. En un entorno artesanal, una práctica muy común es copiar una aplicación previa que se parece a la actual, y a partir de ahí la copia evoluciona de forma totalmente independiente. Aquí el parecido entre las dos aplicaciones sirve para agilizar el desarrollo, pero en ningún caso el mantenimiento. Cada aplicación se mantiene de forma independiente, incluso si este mantenimiento es similar también para ambas. Por tanto, el entorno tradicional tiende a centrarse en el producto, es decir, cada producto tiene su mantenimiento y los equipos humanos también tienden a fragmentarse de esta forma.

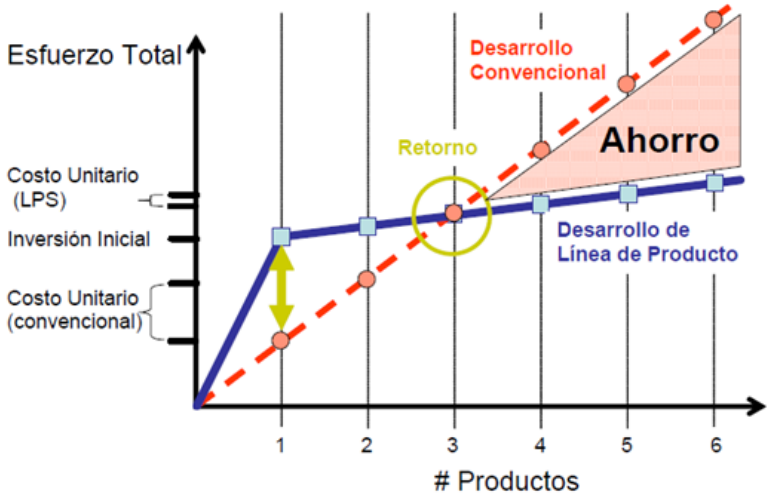

Figura 3. Evolución de costos utilizando desarrollo artesanal y utilizando Línea de Producción de Software.

Por el contrario, un entorno de *LPS* está pensado expresamente para gestionar lo común, y su complementario, lo variable. La reutilización ya no es oportunista, sino planificada, y la incorporación de nuevas variantes se realiza de forma sistemática y controlada. Esto agiliza no sólo el desarrollo del producto y su puesta en el mercado sino también el mantenimiento. Ahora los esfuerzos de mantenimiento realizados en la *LPS* son capitalizados por todos los productos. Un error detectado en un producto puede ser relativamente fácil de corregir en todos los productos de la línea gracias precisamente a la existencia de este marco común que ofrece la *LPS*.

De acuerdo con [NORTHROP 2002], las líneas de productos de software personifican el concepto de la reutilización planificada de forma estratégica, y difieren de la reutilización oportunista del pasado que ha sido ampliamente desacreditada. Las empresas pueden beneficiarse enormemente a través de líneas de productos y la tendencia de la industria hacia las líneas de productos de software parece indiscutible. El Instituto de Ingeniería de Software considera que las líneas de productos de software están aquí para quedarse.

2. PLANTEAMIENTO DEL PROBLEMA

De acuerdo con [PRESSMAN 2005], la industria del software se ha convertido en un factor dominante en la economía del mundo industrializado, donde el programador solitario de la era inicial ha sido sustituido por equipos de especialistas en software, en los que cada uno se enfoca en una parte de la tecnología requerida para desarrollar una aplicación compleja. Sin embargo, persisten las mismas preguntas.

- ¿Por qué tarda tanto la obtención del software terminado?
- ¿Por qué son tan altos los costos de desarrollo del software?
- ¿Por qué es imposible encontrar todos los errores en el software antes de entregarlo a los clientes?

Una situación deseable sería contar de inicio con un ambiente pre-construido del software, integrado con todos los componentes necesarios a utilizar; en este punto, se deben considerar versiones de las bibliotecas, archivos complementarios de un ambiente e incluso estándares de codificación bien definidos.

Se carece de una plataforma que permita administrar líneas de producción de software para la creación de aplicaciones informáticas y que considere lo que se conoce hasta ahora en cuanto a plataformas de línea de producción de software y además la posibilidad de configuración de los activos principales en diferentes lenguajes de programación.

3. OBJETIVOS

Objetivo general

Diseñar, implementar y validar experimentalmente una herramienta de soporte para el desarrollo semi-automatizado de sistemas de información que esté basado en los principios de la línea de productos de software.

Objetivos específicos

1. Diseñar e implementar un entorno gráfico para la administración de línea de productos de software.
2. Diseñar e implementar las componentes variables de la arquitectura de software usando un método jerárquico de características (*features*).
3. Definir por medio de notación *BNF* un lenguaje para la especificación de los componentes variables de las familias de productos.
4. Diseñar e implementar un intérprete para el lenguaje de dominio específico capaz de generar componentes en diferentes lenguajes de programación.
5. Diseñar e implementar un generador automático de documentación de los productos generados mediante la herramienta propuesta.
6. Diseñar e implementar un conjunto de plantillas de software que implementen las componentes estáticas de una familia de productos de software.
7. Desarrollar un caso de prueba en base a la herramienta propuesta.

4. JUSTIFICACIÓN

Se ha elegido este tema de tesis, para aportar alternativas de solución a las problemáticas que se enfrenta la industria del software por la gran diversidad de tecnologías y conocimientos para el desarrollo de sistemas. El interés principal es abonar en una solución tangible, demostrando mediante una herramienta el sentido práctico de la investigación. Como se escribe en [WAYT 1994], el gran desafío es encontrar maneras de cortar los lazos que unen intrínsecamente programas a equipos específicos y a otros programas. Los investigadores están estudiando varios enfoques prometedores, incluyendo un lenguaje común que se podría utilizar para describir las partes de software, los programas que reforman componentes para adaptarse a cualquier entorno, y los componentes que tienen muchas de las características opcionales que el usuario puede activar o desactivar.

El beneficio principal es aproximar la construcción de un software informático de calidad usando un generador de código. Adicionalmente, también hay algunos beneficios secundarios como podrían ser el unificar los esfuerzos técnicos de las personas que integran los equipos, dar certidumbre en los tiempos de entrega del producto, asegurar la calidad de un producto y finalmente, permitir una mejora constante del tipo de aplicación que se puede aproximar.

Al crear una herramienta para una línea de producto de software, se tiene que considerar una solución que se adapte a los distintas familias de sistemas que se desean aproximar. Primero que nada, se debe enfatizar que los productos a considerar, son aquéllos que pueden ser definidos en un mismo dominio, dicho lo anterior, es evidente que no están consideradas aquéllas soluciones que resuelven problemáticas muy especializadas o científicas.

El diseño de la herramienta va enfocado a la administración de varios tipos de productos que se quieran generar, así como también, a la integración de plataformas tecnológicas (lenguajes y estilos de programación). La herramienta cuenta con un administrador de productos que se encarga de reunir las características relevantes de cada producto y una infraestructura con alta adaptabilidad para cumplir con las necesidades de cambios y nuevas tecnologías.

Una de las aportaciones principales de la herramienta, es su capacidad de producción en múltiples tecnologías de software. Para ello, se define un lenguaje de programación de dominio específico que sirve para la transformación de las características de los productos en código fuente, usando un generador de código.

En la tesis también se propone un método de desarrollo para ser llevada por el equipo de trabajo y lograr con éxito la generación de un sistema. También se propone una organización que sería apropiada para los perfiles profesionales que participan en la construcción. El método es necesario, porque con él se obliga a cumplir con los insumos requeridos por la herramienta, ver Figura 4.

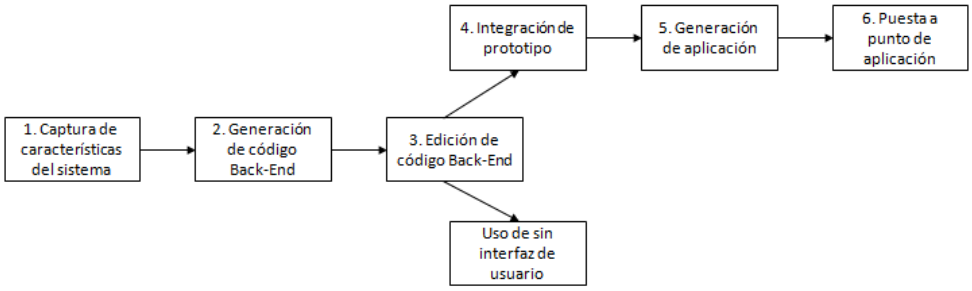

Figura 4. Pasos del método.

El método que se propone se adapta al modelo al proceso de desarrollo secuencial conocido como *waterfall [WIKI 2013]*. Los pasos del método pueden ser usados completamente para generar una aplicación completa o pueden ser usados parcialmente para generar sólo los componentes *Back-End*. Los pasos de integración con el prototipo de la aplicación, son la integración del *Black-End* con la interfaz del usuario.

La tesis es una propuesta tangible de implementación de línea de productos de software, tiene un enfoque de solución que toma en consideración un amplio espectro de tecnologías. En la propuesta se implementan varios diseños principalmente de autores como David L. Parnas, Klaus Pohl, Paul Clements y Charles W. Krueger.

5. ORGANIZACIÓN DEL DOCUMENTO

La tesis está organizada en capítulos y apéndices que resguardan la memoria de la investigación. Los capítulos integran el contenido de la investigación y los apéndices enriquecen temas específicos para dejarlos bien explicados.

Los cinco capítulos en los que se documenta la investigación son la introducción que define la problemática, el estado del arte, el diseño de la herramienta, el caso de estudio y las conclusiones.

CAPÍTULO II

ESTADO DEL ARTE

1. INTRODUCCIÓN

En este capítulo se presenta un análisis del estado del arte y los trabajos relacionados. En el estado del arte se describe un marco de referencia teórico que cuyo objetivo es posicionar la problemática y la solución propuesta. Para ello se buscaron definiciones, investigaciones y productos comerciales relacionados de algún modo con la solución propuesta. La presentación está organizada de la parte más general hasta la más particular.

Iniciamos considerando las definiciones de ingeniería de software, que es el marco general al cual pertenece el desarrollo de sistemas informáticos. El desarrollo de sistemas es estudiado desde la ingeniería de dominio y desde la ingeniería de aplicación, la forma de abordarlo, depende de si se crean varias aplicaciones del mismo dominio o si se crea solo una aplicación. También explicamos un poco la Ingeniería de Requerimientos, debido a que sus paradigmas son ampliados para integrar un análisis y diseño que nos lleve a la creación de productos de calidad, ver Figura 5.

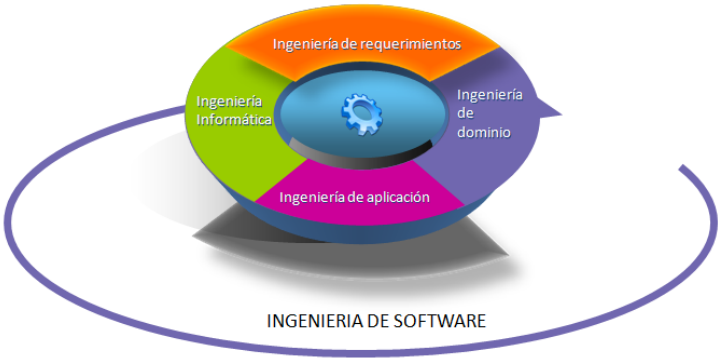

Figura 5. Marco teórico de Líneas de Productos de Software.

Esta investigación tiene por objetivo diseñar y probar una herramienta que semi-automatiza la construcción de aplicaciones informáticas y que pertenecen a un dominio común. La automatización lograda, tiene las características de una línea de producción de software, ya que la gran mayoría de las aplicaciones tienen características comunes y pertenecen a una misma familia.

Una vez que tenemos nuestro marco teórico, entramos a detallar los paradigmas de programación generativa, lenguajes de dominio específico y meta-programación. Estos tres paradigmas serán parte fundamental en el diseño e implementación de nuestra herramienta.

La explicación de lenguajes de dominio específico tiene especial importancia, ya que con el diseño e incorporación de un lenguaje de este tipo, logramos independizar la generación de componentes de los distintos lenguajes de programación.

Actualmente podemos encontrar enfoques de solución similares al propuesto en esta tesis, incluso ya hay herramientas comerciales en el mercado. En la segunda parte de este documento, se describen estos trabajos y herramientas relacionadas al mismo enfoque de generación de código.

2. INGENIERÍA DE SOFTWARE

De acuerdo con la definición proporcionada en [IEEE 1990, pp.67], la ingeniería de software es la aplicación de un enfoque sistemático, disciplinado y cuantificable para desarrollar, operar y mantener el software.

Según la definición del IEEE citada por [Lewis 1994], el *"software es la suma total de los programas de computadora, procedimientos, reglas, la documentación asociada y los datos que pertenecen a un sistema de cómputo"*.

Las definiciones nos refieren que la ingeniería de software incluye un conjunto de conceptos, metodologías, técnicas de medición y herramientas para tener un software de calidad. La disciplina tiene lo que se necesita para el tratamiento asertivo y comprobable de cada tipo de software, así como de su proceso de creación. Esta investigación se concentra sólo en el proceso de creación, y propone un enfoque que permite acelerarlo.

El proceso de creación de software está considerado por la ingeniería de software y para ello, define sus principios y métodos para lograr un buen término del proceso, además, la disciplina provee de herramientas para la construcción de un software de calidad con un presupuesto limitado y con una fecha de término.

La gran mayoría de métodos en el desarrollo de software refieren una relación preponderante con las personas que participan en el proceso de creación. Los métodos incluyen aspectos de gestión, control y auditoría para mitigar circunstancias normales o imponderables, en un contexto de constantes requerimientos de cambio.

El proceso de desarrollo de software trata con los requerimientos del sistema, las especificaciones del sistema y un diseño detallado de su arquitectura. Adicionalmente, los sistemas necesitan ser verificados y validados. Los requerimientos toman tal

relevancia, que sus principios, métodos y herramientas forman parte de la ingeniería de requerimientos.

Hasta hace poco tiempo, para crear un sistema, bastaba con reutilizar el código, copiando y pegando los componentes necesarios. Las variantes del software que se encontraban se trataban una a una. Esta forma de desarrollar código fue ampliamente utilizada usando sistemas tipo, y adaptando las particularidades en fases posteriores [KLAUS GUNTER FRANK, 2005, pp.13]. Esta forma de construcción no tenía gran ciencia, sin embargo, no todas las particularidades son exactamente pequeñas o todos los sistemas son del mismo tipo. Además, los expertos tenían que ser especialistas en el tipo para encontrar una forma rápida y elegante de adaptar las variantes, en caso contrario, las adaptaciones podían ser muy costosas y de alto riesgo.

Hoy en día, la situación del software ha cambiado, muchos de nuestros sistemas de software se están haciendo intensivos, no solo por la variabilidad en su diseño, sino también por la integración de nuevas funciones para tenerlos al día. La cantidad de software incluido está creciendo, y la cantidad de variabilidad está creciendo aún más rápido. Lo anterior trajo como consecuencia una fuerte necesidad para adaptar la ingeniería de línea de productos en software del mismo dominio [KLAUS GUNTER FRANK, 2005, pp.14].

El desarrollo de software basado en líneas de productos, tiene una estrecha relación con la ingeniería de dominio y la ingeniería de aplicación. *La ingeniería de dominio* se centra en el desarrollo de elementos reutilizables que formarán una familia de productos, y la *ingeniería de aplicación* se orienta hacia la construcción o desarrollo de productos individuales pertenecientes a la familia de productos, y que satisfacen un conjunto de requisitos y restricciones expresados por un usuario específico, ver Figura 6.

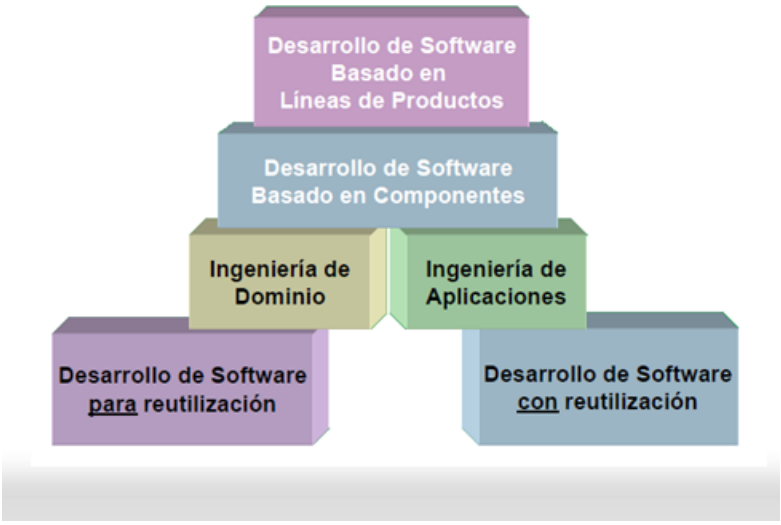

Figura 6. Reutilización de Software.

El proceso de desarrollo de software, requiere por un lado, un conjunto de conceptos y por el otro, una metodología. A este proceso se le llama el ciclo de vida del software. En esta investigación se propone el uso de una metodología específica para la construcción de una serie de productos de software de una misma familia.

2.1. Ciclo de Desarrollo de Software

De acuerdo con la definición presentada en [IEEE 1990, pp.67], el ciclo de desarrollo de software es el período de tiempo que inicia con la decisión de desarrollar un producto de software y termina cuando el software es entregado. El ciclo normalmente incluye una fase de requerimientos, fase de diseño, fase de implementación, fase de pruebas y en algunas ocasiones, la fase de instalación y pruebas, ver Figura 7.

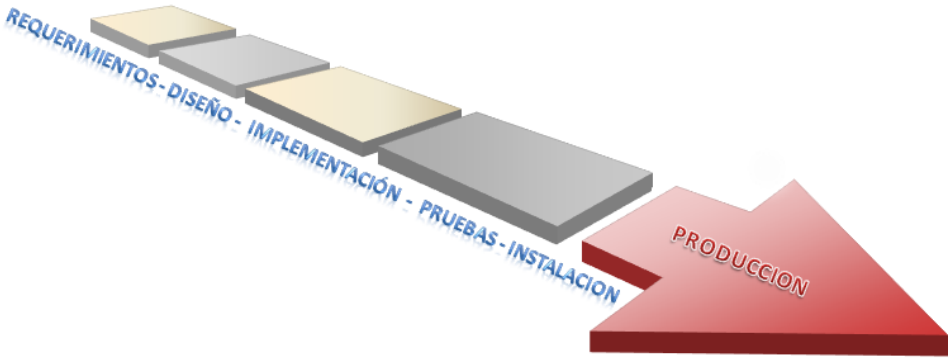

Figura 7. Ciclo de desarrollo de software.

A decir de la definición, el ciclo comprende todas las actividades que suceden desde que identificamos la necesidad de contar con un software y se inicia su construcción. El ciclo de vida tiene por objetivo, detectar que los errores se disminuyan o se encuentren lo antes posible, para que el equipo de trabajo se concentre en la calidad del software, en los plazos de implementación y en los costos asociados.

2.2. Procesos de Desarrollo de Software

De acuerdo con la definición presentada en [IEEE 1990, pp.67], el proceso de desarrollo de software es un procedimiento por el cual se transforman las necesidades del usuario, en un producto de software. El proceso incluye la transición de los requerimientos del usuario en requerimientos de software, la transformación de requerimientos en un diseño, implementación del diseño en código, pruebas del código, y algunas veces, instalación y puesta a punto del software para su uso operacional, ver Figura 8.

De acuerdo al diccionario Oxford, un procedimiento es un modo establecido u oficial de hacer algo. Las organizaciones tienen como un principio fundamental, documentar todos los procesos.

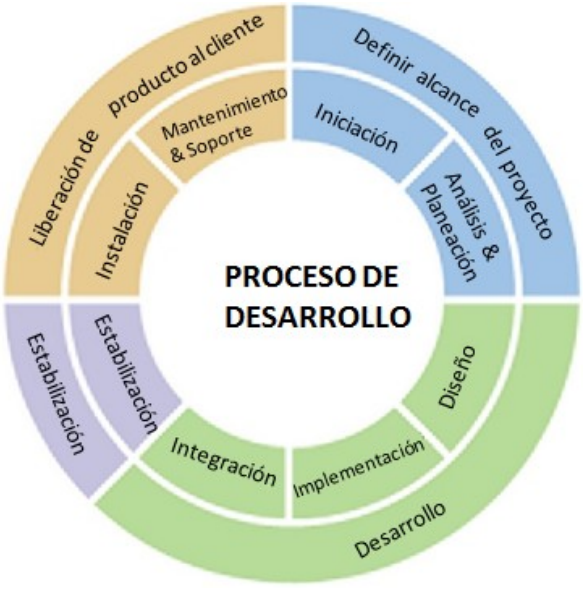

Figura 8. Procesos de desarrollo de software.

INICIALIZACIÓN

La inicialización del proyecto, contiene la especificación de requisitos de software y la definición de los casos de uso, aplicando un análisis y diseño a bajo nivel.

ANÁLISIS Y PLANEACIÓN

El análisis, consiste en organizar la información recogida de manera que pueda interpretarse antes de proceder al diseño del sistema. Esta interpretación se presenta como el documento de requisitos el cual es una descripción clara y precisa de los requisitos del área de negocio.

La planificación del proyecto presenta la estimación del costo del proyecto, el cálculo del tamaño y la cantidad de días para el desarrollo del proyecto. De igual forma, incluye la planificación para especificar un horario de trabajo con el total de los días obtenidos en la estimación.

DISEÑO

Consiste en describir como el sistema va a satisfacer los requerimientos. Esta etapa a menudo tiene diferentes niveles de detalle. Los niveles más altos de detalle generalmente describen los componentes o módulos que formarán el software a ser producido. Los niveles más bajos, describen, con mucho detalle, cada módulo que contendrá el sistema.

IMPLEMENTACIÓN

Es la fase donde el software a ser desarrollado se codifica. Dependiendo del tamaño del proyecto. La programación puede ser distribuida entre distintos programadores o grupos de programadores. Cada uno se concentrará en la construcción y prueba de una parte del software, a menudo un subsistema. Las pruebas, en general, tienen por objetivo asegurar que todas las funciones están correctamente implementadas dentro del sistema.

INTEGRACIÓN

Es la fase donde todos los subsistemas codificados independientemente se juntan. Cada sección es enlazada con otra y, entonces, probada. Este proceso se repite hasta que se han agregado todos los módulos y el sistema se prueba como un todo.

ESTABILIZACIÓN

Una vez que el sistema ha sido integrado, comienza esta etapa. Es donde es probado para verificar que el sistema es consistente con la definición de requerimientos y la especificación funcional. Por otro lado, la estabilización consiste en una serie de actividades que aseguran que el software implementa correctamente una función específica. Al finalizar esta etapa, el sistema ya puede ser instalado en ambiente de explotación.

INSTALACIÓN

Es el proceso por el cual nuevos programas son transferidos a un computador con el fin de ser configurados y preparados para ser ejecutados en el sistema informático, para cumplir la función para la cual fueron desarrollados.

MANTENIMIENTO Y SOPORTE

El mantenimiento ocurre cuando existe algún problema dentro de un sistema existente, e involucraría la corrección de errores que no fueron descubiertos en las fases de prueba, mejoras en la implementación de las unidades del sistema y cambios para que responda a los nuevos requerimientos. Los mantenimientos se pueden clasificar en correctivos, adaptativos, perfectivos y preventivos.

2.3. Ingeniería de Requerimientos

De acuerdo con definiciones encontradas en [IEEE 1990, pp.62], un requerimiento puede definirse de las siguientes formas. (1) Un requerimiento es una condición o capacidad necesaria por un usuario para resolver un problema o alcanzar un objetivo. (2) Una condición o capacidad que debe conocerse y procesarse por una serie de componentes o sistema para satisfacer un contrato, estándar, especificación, u otra clase de documentación formalmente impuesta.

La definición más amplia de ingeniería de requerimientos es una que llega de un documento de estrategia de software *DoD* (*Department of Defense*) de 1991 y citado por [ELIZABETH KEN JEREMY 2011, pp.8]. En ese documento se define a la *ingeniería de requerimientos* como todo lo que incluye el ciclo de vida de las actividades dedicadas a identificar los requerimientos del usuario, analizar los requerimientos para derivar en requerimientos adicionales, documentar los requerimientos como una especificación, y

validar los requerimientos documentados contra las necesidades del usuario. Lo anterior, considerando también los procesos que soportan estas actividades.

Mientras esta definición cubre la identificación, análisis, desarrollo y validación de requerimientos, también omite mencionar la importancia que juegan los requerimientos en la aceptación y verificación de la solución. Una definición más actualizada y contextualizada en la ingeniería de software es la presentada en [ZAVE 1997], que dice que la ingeniería de requerimientos es la rama de la ingeniería de software interesada en los objetivos del mundo real para funciones y limitaciones en los sistemas de software. También se interesa en la relación de estos factores con las especificaciones precisas de la conducta del software, y su evolución en el tiempo y a través de familias de software.

Los requerimientos se documentan en una *especificación de requerimiento de software* (*SRS,* por sus siglas en inglés). La definición de una *SRS* la podemos encontrar en [IEEE 1998, pp.3], donde dice que la *SRS* es una especificación para un software en particular, programa, o un conjunto de programas para ejecutar ciertas funciones en un medio ambiente específico.

Además en [IEEE 1998, pp.3-4], nos definen las cuestiones básicas que deben ser abordadas en un *SRS.*

a. *Funcionalidad*. ¿Qué se supone que hace el software?
b. *Interfaz externa*. ¿Cómo hace el software para interactuar con personas, hardware de sistemas, otro hardware y otro software?
c. *Desempeño*. ¿Cuál es la velocidad, disponibilidad, tiempo de respuesta, tiempo de recuperación de varias funciones del software, etc.?
d. *Atributos*. ¿Cuáles son las condiciones de la portabilidad, corrección, mantenimiento, seguridad, etc.?
e. *Limitaciones de diseño impuestas en una implementación*. ¿Hay algún estándar que se requiere considerar, lenguaje de implementación, políticas de integridad de base de datos, limitaciones de recursos, ambientes operativos, etc.?

Las características que debe poseer un buen *SRS* son las siguientes. Deben ser *correctas, sin ambigüedad, completas, consistentes, calificadas por importancia y/o estabilidad, verificables, modificables y rastreables.*

Una consideración importante para incluir en un *SRS* son los prototipos. Un prototipo es un tipo preliminar, forma, o instancia de un sistema que sirve como un modelo para etapas posteriores o para la versión final y completa de un sistema [IEEE 1990, pp.60].

De acuerdo con [IEEE 1998, pp.9], los prototipos son útiles por las siguientes razones:

a. El cliente puede tener más probabilidades de ver el prototipo y reaccionar a él que leer el *SRS* y reaccionar a él. De este modo, el prototipo provee una rápida retroalimentación.
b. El prototipo muestra aspectos no previstos de la conducta de sistemas. Así, se producen no sólo respuestas, sino también nuevas preguntas. Esto ayuda a concluir un *SRS*.
c. Un *SRS* basado en un prototipo tiende a sufrir menos cambios durante el desarrollo, lo que reduce el tiempo de desarrollo.

Los prototipos pueden ser utilizados como una técnica dentro de una fase en un modelo de desarrollo o como un modelo por si mismo. El caso del modelo espiral, considera el prototipo como un mecanismo clave en el éxito del modelo [BARRY 2001]. El modelo de prototipo rápido, se basa fundamentalmente en prototipos.

2.4. Líneas de Productos de Software

Los conceptos fundamentales de familia de programas que llevaron a la creación del paradigma de Línea de Productos de Software, fueron abordados por primera vez por David L. Parnas en [PARNAS 1976], el cual se basó en el artículo de Edsger W. Dijkstra de programación estructurada de 1970.

En [PARNAS 1976], encontramos que para constituir una familia de productos de software primero se deben estudiar las propiedades comunes y después las propiedades especiales de cada miembro. David L. Parnas, también escribe que para el desarrollo de programas, es mejor utilizar un método secuencial completo y después de tener una versión trabajando, es posible modificarlo para ser un nuevo miembro de la familia.

También en el mismo artículo, se define un método de representación de desarrollo de miembros de una familia llamado "decisiones abstractas", el cual propone un desarrollo de versiones que son modelados como un árbol con ramificaciones (Figura 9), donde cada ramificación es considerada como subfamilia, y donde el segmento de nodos secuenciales (antes del primer nodo de decisión) es considerado como el desarrollo de un miembro con un método secuencia tradicional.

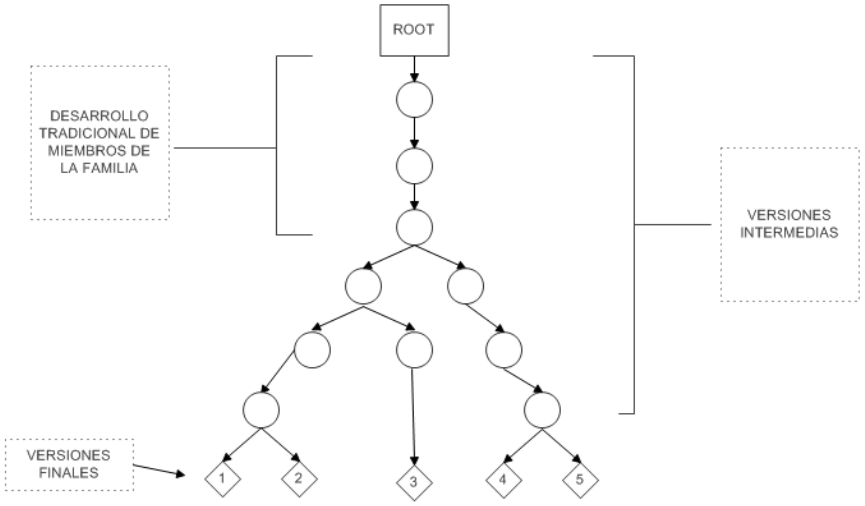

Figura 9. Representación del desarrollo de programa usando "decisiones abstractas".

2.4.1. Línea de Producción

La forma en que se producen los bienes ha cambiado significativamente en el curso de tiempo. Anteriormente, los bienes fueron hechos a mano para contados clientes individuales. Poco a poco aumentó el número de personas que podían permitirse el lujo de comprar productos. En el ámbito de los automóviles, esto condujo a la invención de la línea de producción para Ford, lo que permitió la producción mucho más barata para un mercado de masas, comparado con la creación de cada producto de forma artesanal. Sin embargo, la línea de producción redujo las posibilidades de diversificación.

Por un tiempo, los clientes estaban satisfechos con los productos masivos estandarizados – pero no todas las personas quieren el mismo tipo de coche para cualquier propósito. Mientras algunos coches son utilizados para que viajen pocas personas, otros son diseñados para que viajen familias numerosas. Algunos coches son utilizados por las personas que viven en las ciudades, otros en el campo. La gente quiere tener un segundo carro o uno mejor que sus vecinos. Como consecuencia, la industria enfrentó una creciente demanda de productos individualizados. Este fue el inicio de la personalización en masa, lo que significa tener en cuenta las necesidades de los clientes y darles lo que quieren.

Para el cliente, la personalización en masa significa la posibilidad de tener un producto individualizado. Para la empresa, la personalización en masa se traduce en mayor inversión tecnológica, que conduce a costos de producción más altos y/o a menores

márgenes de ganancia para los inversionistas. Ambos efectos son indeseables. Así que, muchas empresas, sobre todo en la industria del automóvil, comenzaron a presentar *plataformas* comunes para los diferentes tipos de vehículos mediante la planificación de antemano, de qué partes se utilizan en diferentes tipos de coches [KLAUS GUNTER FRANK, 2005, pp.4], ver Figura 10.

Figura 10. Línea de producción con personalización masiva.

Los cambios profundos en las prácticas de ingeniería generalmente no se inician sin una sólida justificación económica. Una razón fundamental para la introducción de la línea de productos de ingeniería es la reducción de costos. Cuando los componentes comunes se reutilizan en varios tipos diferentes de sistemas, esto implica una reducción de costos para cada sistema [KLAUS GUNTER FRANK, 2005, pp.9].

Las motivaciones para la ingeniería de productos son:

1. Reducción de costos de desarrollo
2. Incremento en la calidad
3. Reducción de tiempo para el mercado

2.4.2. Línea de Productos de Software

De acuerdo a la definición [KLAUS GUNTER FRANK, 2005, pp.14*], la ingeniería de línea de productos de software es un paradigma para desarrollar aplicaciones de software(sistemas de software intensivos y productos de software) usando plataformas y customización en serie.*

La diferencia clave entre el desarrollo tradicional de sistemas únicos y la ingeniería de línea de productos de software es un cambio fundamental de enfoque. Ir desde los proyectos y sistemas individuales a una línea de producto. Este cambio especialmente

implica un cambio en la estrategia, es decir, en la visión ad-hoc después del contrato a una vista estratégica en el área de los negocios [FRANK KLAUS EELCO 2007, pp.6].

Las motivaciones para la ingeniería de productos de software son:

1. Reducción de esfuerzos de mantenimiento.
2. Copiado con evolución.
3. Copiado con complejidad.
4. Mejora en la estimación de costos.
5. Beneficios para los clientes.

Para [HEYMANS TRIGAUX 2003], el propósito para la reducción de tiempos y costos de producción, es incrementar la calidad del software por medio del reúso de componentes, los cuales ya han sido probados y asegurados. Estos objetivos se pueden alcanzar dejando en los artefactos de desarrollo comunes los documentos de requerimiento, diagramas de conceptualización, arquitectura, código (componentes reusables), procedimientos de pruebas, procedimientos de mantenimiento, etcétera.

Los mecanismos que respaldan la teoría de Línea de Productos de Software están divididos en tres procesos paralelos principales. El desarrollo de activos núcleo, el desarrollo de producto y la administración. De hecho, los productos y activos núcleo son influenciados mutuamente y evolucionan en paralelo durante el ciclo de vida [NORTHROP 2002].

2.4.3. Administración de la Variabilidad

La ingeniería de línea de productos de software optimiza el desarrollo de sistemas individuales aprovechando sus características comunes y administrando sus diferencias de manera semántica [CLEMENTS NORTHROP 2001].

Para [OUALI KRAIEM GHEZALA 2012], estas diferencias son llamadas variabilidades. En la Ingeniería de Línea de Producto de Software, se pueden distinguir dos tipos de variabilidad. La variabilidad de software y la variabilidad de línea de producto. La variabilidad de Software se refiere a la habilidad de un sistema de software para ser eficientemente extendido, cambiado, customizado o configurado para usarse en un contexto particular. Mientras que la variabilidad de línea de producto describe la variación en el sistema que pertenece a una línea de producto.

La variabilidad es usualmente descrita con puntos de variación y variantes. Un punto de variación localiza una variabilidad y su enlace que refieren a varias variantes. Cada variante corresponde a una alternativa diseñada para realizar una variabilidad particular y de este modo, enlazar a un camino concreto. En otras palabras, el punto de variación apunta a las variantes y determina las características (*atributos*) de la variabilidad [HEYMANS TRIGAUX 2003].

Cuando administramos la variabilidad en una línea de producto, necesitamos distinguir tres tipos principales [FRANK KLAUS EELCO 2007].

1. Características comunes: Una característica (funcional o no-funcional) puede ser común a todos los productos en la línea de producción. Esta es implementada como parte de la plataforma.

2. Variabilidad: Una características que puede ser común para algunos productos, pero no para todos. Debe ser explícitamente modelada como una posible variabilidad y debe ser implementada de un modo que permita tenerla solamente en productos seleccionados.

3. Específicas a un producto: Una característica que puede ser parte sólo de un producto – al menos para un futuro previsible. Tales especialidades no son a menudo requeridas por el mercado per-se, sino que se deben a las preocupaciones de clientes particulares.

Durante el ciclo de vida de la línea de productos, una variabilidad específica puede cambiar de tipo. Por ejemplo, una característica específica del producto puede convertirse en una variabilidad. Por otro lado y del mismo modo, una característica común puede convertirse en una variabilidad.

2.4.4. Plataforma de Software

El enfoque de plataforma, es cualquier base de tecnologías sobre las cuales otras tecnologías o procesos son construidos. La combinación de personalización masiva y una plataforma común, nos permite utilizar una base común de tecnología y al mismo tiempo sacar productos acordes a los deseos de los clientes [FRANK KLAUS EELCO 2007, pp.6-7].

El desarrollo de aplicaciones usando una *plataforma de software*, significa planificar proactivamente para la reutilización. Construir aplicaciones con personalización masiva,

significa emplear el concepto de gestión de la variabilidad. La gestión de variabilidad tiene gran impacto en la forma en que se desarrolla el software. Sin embargo, si para estos cambios no se utiliza una *plataforma*, a menudo se corrompe la estructura original del software y obstaculizan los aspectos de calidad [FRANK KLAUS EELCO 2007, pp.14].

De acuerdo a la definición presentada en [KLAUS GUNTER FRANK, 2005, pp.15], la plataforma de software es un conjunto de subsistemas e interfaces de software que forman una estructura común desde donde pueden ser eficientemente desarrollados y producidos un grupo de productos derivados.

Una plataforma de software que implemente el paradigma de línea de producción de software, se debe separar en dos procesos. *Ingeniería de dominio* e *Ingeniería de aplicación, ver Figura 11*.

El proceso de *ingeniería de dominio*, es responsable de establecer la plataforma reutilizable y de este modo, define la parte común y variable de una línea de producto. El proceso de *ingeniería de aplicación*, es responsable de implementar las necesidades específicas de la aplicación [KLAUS GUNTER FRANK, 2005, pp.21].

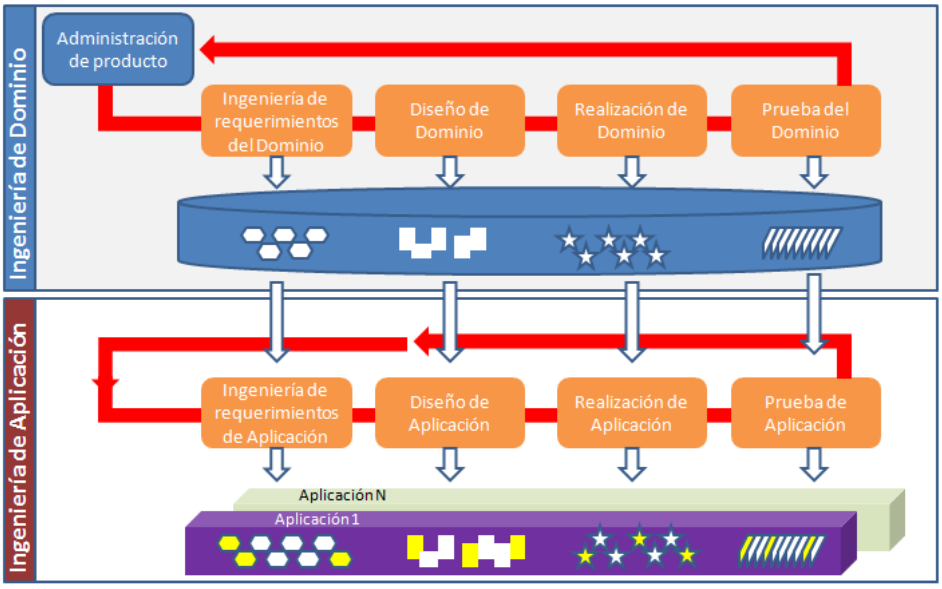

Figura 11. Diseño de la plataforma de software.

Es importante tener en cuenta que ni los sub-procesos del dominio y de aplicación, ni sus actividades, tienen que realizarse en un orden secuencial.

La administración de producto, trata con el aspecto económico de la línea de producto de software y en particular de la estrategia de mercado. Su principal objetivo, es administrar el portafolio de productos de la compañía o unidad de negocio.

2.5. Ingeniería de Dominio

El proceso general de las líneas de productos está basado en la reusabilidad de requerimientos, arquitectura y componentes. La Ingeniería de Dominio o desarrollo para reusar, consiste en el desarrollo de activos núcleo a través de los procesos de análisis, diseño e implementación de dominio [HEYMANS TRIGAUX 2003].

De acuerdo a la definición presentada en [HARSU 2002], el propósito de la ingeniería de dominio es proporcionar la reutilización entre aplicaciones similares (por ejemplo, una familia de aplicaciones). La ingeniería de dominio, es un proceso sistemático para proporcionar una arquitectura de núcleo común para estas aplicaciones.

El dominio para la ingeniería de software como lo define *Wikipedia* es un campo de estudio que define un conjunto de requerimientos comunes, terminología, y funcionalidad para cualquier programa de software construido para solucionar un problema en el área de programación de computadoras, conocida como ingeniería de dominio.

Podemos encontrar otra definición en [KLAUS GUNTER FRANK, 2005, pp.21] donde la ingeniería de dominio se describe como el proceso de ingeniería de línea de productos de software en el cual son definidas y realizadas las partes comunes y variables de la línea de productos.

De aquí podemos resumir que un dominio es un área de aplicación de productos de software, que están centrados sobre una misma área de conocimiento, que es factible de producirlo en línea y que puede dividirse en sub dominios comunes y variables, ver Figura 12.

El manejo de la variabilidad es tratado en todo el ciclo de vida del software. Este inicia con la definición de alcance inicial, se presenta en toda la etapa de implementación y pruebas y finalmente hasta en la evolución. Como tal, la variabilidad es relevante para todos los productos a lo largo del desarrollo de software [FRANK KLAUS EELCO 2007, pp.9].

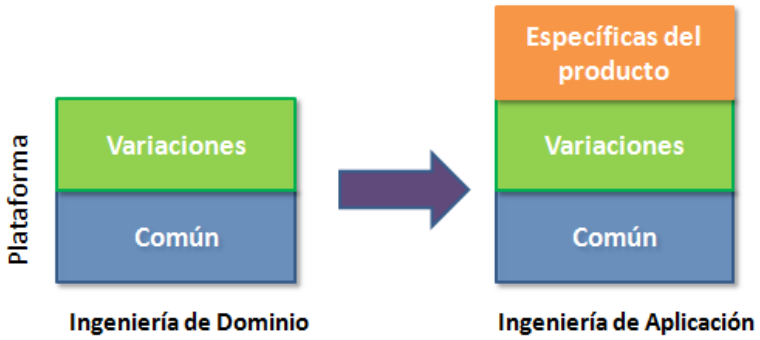

Figura 12. Relación de los diferentes tipos de variabilidad.

La mayoría de los enfoques modernos usan las *features* (características) como los conceptos básicos de representación de la variabilidad. Sin embargo, existen varias interpretaciones del término *features*. El punto más importante es que las características de los productos que los diferencía de otros productos son la esencia para la representación. La mayoría de los enfoques modernos soportan la caracterización de la variabilidad por medio de características que pueden representarse en varias vistas [FRANK KLAUS EELCO 2007, pp.9].

2.6. Ingeniería de Aplicación

En [KLAUS GUNTER FRANK, 2005, pp.21] la *ingeniería de aplicación*, se define como el proceso de ingeniería de línea de productos de software en el cual las aplicaciones de la línea de producto son construidos por artefactos que reúsan el dominio y explotan la variabilidad de la línea de producto.

Otra definición la encontramos en [KRZYSZTOF 1998, pp.33], donde la ingeniería de aplicación, se establece como el proceso para construir sistemas basados en el resultado de la ingeniería de dominio. Durante el análisis de requerimientos para un nuevo sistema, tomamos ventaja del modelo de dominio existente y seleccionamos los requerimientos (features) desde el modelo de dominio el cual resuelve las necesidades del cliente. Por supuesto, los requerimientos nuevos del cliente que no se encuentran en el modelo de dominio requieren un desarrollo específico. Finalmente, se ensambla la aplicación con los componentes reusables y los componentes desarrollados conforme a la arquitectura reusable, o, idealmente, le dejamos al generador hacer el trabajo.

Los objetivos clave de los procesos de ingeniería de aplicación son los siguientes.

- Lograr la reutilización tan alta como sea posible de los productos del dominio cuando se defina y desarrolle una aplicación de línea de producto.
- Explotar la parte común y variable de la línea de producto de software durante el desarrollo de aplicaciones de línea de producto.
- Documentar la plataforma de software. Por ejemplo los requerimientos, arquitectura, componentes y pruebas, y relacionarlos a la plataforma de dominio.
- Asociar la variabilidad de acuerdo a las necesidades de la aplicación de los requerimientos sobre la arquitectura, a componentes, y casos de prueba.
- Estimar el impacto de las diferencias entre requerimientos de aplicación y dominio sobre la arquitectura, componentes, y pruebas.

2.7. Programación Generativa

K. Czarnecki define la Programación Generativa (*GP,* por sus siglas en inglés) como la que trata del diseño e implementación de módulos de software los cuales pueden ser combinados para generar sistemas especializados y altamente optimizados cumpliendo con requerimientos específicos [KRZYSZTOF 1998, pp.7].

La programación generativa considera que el aumento de la productividad pasa por elevar el nivel de abstracción de los lenguajes de programación, haciendo el uso de especificaciones, que son representadas a través de modelos. Un aspecto clave de este paradigma es la traducción automática de los modelos a código ejecutable, y uno de los enfoques utilizados es aplicar las transformaciones sobre la especificación de los productos. La implementación de estas transformaciones es un proceso bastante complejo y suelen estar soportadas por lenguajes que carecen las prestaciones típicas de los lenguajes de programación.

Los objetivos de la *GP* según [KRZYSZTOF 1998, pp.7] son los siguientes.

a. Disminuir la brecha conceptual entre el código del programa y los conceptos de dominio.
b. Lograr una alta reusabilidad y adaptabilidad.
c. Simplificar la gestión de muchas variantes de un componente.
d. Incrementar la eficiencia en espacio y tiempo de ejecución.

Para alcanzar estos objetivos, *GP* emplea los siguientes principios:

- Separación de preocupaciones. Se refiere a la importancia de tratar con una cuestión importante a la vez. Para evitar que el código del programa trate con muchos problemas simultáneamente, la programación generativa tiene por objetivo separar cada problema en un conjunto de código distinto. Estas piezas de código, son combinadas para generar un componente necesario.
- Configuración de diferencias. Tal y como en la programación genérica, los parámetros de configuración nos permite compactar la representación de familia de componentes.
- Análisis y modelado de dependencias e interacciones. No todas las combinaciones de parámetros son usualmente válidas, y los valores de algunos parámetros podrían implicar el valor de algunos otros parámetros. Las dependencias son referidas como el conocimiento de configuración horizontal, ya que este ocurre entre parámetros en un nivel de abstracción.
- Separación del espacio de problema del espacio de solución. El espacio de problema consiste de las abstracciones de un dominio específico con el cual deberían interactuar los programadores de la aplicación. Mientras que el espacio de solución contiene la implementación de componentes.
- Eliminación de sobrecarga y optimización del desempeño de un dominio específico. Generando componentes estáticamente (en tiempo de compilación), es posible encontrar mucha sobre carga debido a código no utilizado, entonces pueden ser eliminados los códigos de *depuración* y los niveles innecesarios no utilizados.

2.8. Lenguaje de Dominio Específico

En el desarrollo de software, un lenguaje específico de dominio (*DSL,* por sus siglas en inglés) es un lenguaje de programación dedicado a un problema de dominio en particular, o una técnica de representación o resolución de problemas específica. Este concepto no es nuevo, ya que desde siempre existieron lenguajes de programación de propósito específico.

De acuerdo con K. Czarnecki, un lenguaje de dominio específico, provee características de un lenguaje especializado que incrementa el nivel de abstracción para un dominio de problema particular, el lenguaje permite trabajar estrechamente con los conceptos del dominio(intencionalmente), pero con un costo de generalidad del lenguaje [KRZYSZTOF

1998, pp.8]. Lo opuesto a un lenguaje específico de dominio son los lenguajes de programación de propósito general, como C o Java.

Como ejemplos de *DSL* podemos mencionar a las fórmulas y macros de las planillas de cálculo, las expresiones regulares de ciertas utilidades, Csound (un lenguaje para crear archivos de audio), y más.

Crear un *DSL* (con el software que lo soporte) puede valer la pena si el lenguaje permite expresar tipos de problemas y soluciones particulares que los lenguajes pre-existentes no pueden modelar tan fácilmente. Los *DSL* son lenguajes con objetivos muy específicos, tanto en su diseño como en su implementación. Un *DSL* puede ser tanto un lenguaje de diagramación visual (como el creado por *Generic Eclipse Modeling System*), o lenguajes textuales. Por ejemplo, la utilidad de línea de comandos *grep* tiene una sintaxis de expresiones regulares para buscar líneas de texto.

La línea que divide a los *DSL* y a los lenguajes de *scripting* es bastante difusa, pero en general los *DSL* no tienen funciones de bajo nivel para acceder al sistema de archivos, ni control de interprocesos ni otras características que sí poseen los lenguajes completos. Muchos *DSL* no se compilan a código ejecutable, sino a diferentes objetos como por ejemplo, GraphViz exporta a PostCript, GIF, JPEG, etc., Csound compila a archivos de audio, y el lenguaje POV compila hacia archivos gráficos.

El lenguaje *SQL* es un caso interesante ya que podría considerarse un *DSL* porque es específico para un dominio (en este caso, acceder a las bases de datos relacionales), y se suele utilizar dentro de otra aplicación. Sin embargo, *SQL* tiene más palabras clave y funciones que muchos lenguajes de *scripting*, y se suele pensar en *SQL* como en un lenguaje completo (quizás por la enorme presencia de manipulación de bases de datos en los programas, y por la gran cantidad de experiencia que se necesita para ser experto en este lenguaje).

Existen riesgos y oportunidades al adoptar un *DSL* para el desarrollo de aplicaciones. Un *DSL* bien diseñado va a lograr el balance apropiado entre ambas.

Los *DSL* tienen metas de diseño importantes que contrastan con aquellas de los lenguajes de propósito general. En concreto:

- los *DLS* tienen menos alcance.
- los *DSL* son mucho más expresivos dentro de su dominio.

VENTAJAS DE LOS DSL

Los *DSL* permiten expresar soluciones usando los términos y el nivel de abstracción apropiado para el dominio del problema. En consecuencia, los mismos expertos de dominio pueden comprender, validar, modificar y a menudo desarrollar programas en *DSL*.

ES CÓDIGO AUTO-DOCUMENTADO.

Los *DSL* mejoran la calidad, productividad, confianza, mantenimiento, portabilidad y reusabilidad de las aplicaciones.

Los *DSL* permiten validaciones a nivel del dominio. Mientras las construcciones del lenguaje estén correctas, cualquier sentencia escrita puede considerarse correcta.

DESVENTAJAS DE LOS DSL

Por otro lado, los *DSLs* también tienen desventajas, entre ellas podemos mencionar las siguientes.

- El costo de aprender un nuevo lenguaje contra su aplicación limitada.
- El costo de diseñar, implementar y mantener un *DSL* y las herramientas para trabajar con él.
- Encontrar, establecer y mantener el alcance adecuado entre las necesidades específicas y solicitudes generales.
- Dificultad para balancear las ventajas y desventajas entre las construcciones de los *DSL* y de los lenguajes de propósito general.
- Potencial pérdida de eficiencia y rendimiento en comparación con el software escrito "a mano".

2.9. Sistemas de Meta Programación

La **meta-programación** consiste en escribir programas que escriben o manipulan otros programas (o a sí mismos), o que hacen en tiempo de compilación parte del trabajo que, de otra forma, se haría en tiempo de ejecución. Esto permite al programador ahorrar tiempo en la producción de código.

Un ejemplo sencillo de un meta-programa es el *script* de Bash de la Figura 13. Este *script* genera un nuevo programa que imprime por pantalla los números 1 a 992. Esto

es sólo una muestra de como usar código para escribir más código, no la forma más eficiente de imprimir una lista de números. En cualquier caso, un buen programador puede escribir y ejecutar este meta-programa en apenas un par de minutos, y habrá generado exactamente 1000 líneas de código en esa cantidad de tiempo.

```
#!/bin/bash
# metaprogram
echo '#!/bin/bash' >program
for ((I=1; I<=992; I++)); do
   echo "echo $I" >>program
done
chmod +x program
```

Figura 13. Meta-programación.

La herramienta de meta-programación más común es el compilador, el cual permite al programador escribir un programa relativamente corto en un lenguaje de alto nivel para, posteriormente, escribir un programa equivalente en lenguaje ensamblador o lenguaje máquina. Esto, por lo general, significa un buen ahorro de tiempo si se compara con la posibilidad de escribir el programa en lenguaje máquina de forma directa.

Otro ejemplo bastante común de meta-programación se puede encontrar en el uso de *Lex* (véase también: *Flex*) y *Yacc* (véase también: bison), que son usados para generar compiladores e intérpretes.

3. TRABAJOS RELACIONADOS

3.1. Trabajos de Investigación Relacionados

3.1.1. S.P.L.O.T. – Software Product Lines Online Tools

Es un sistema de configuración y razonamiento basado en web para las líneas de productos de software. El sistema se beneficia de técnicas. de razonamiento maduro basado en la lógica como los solucionadores *SAT* (*boolean SATisfiability problem*) y diagramas de decisión binarios, para proveer a las *LPS* de investigadores y profesionales con servicios de configuración interactivo y razonamiento eficiente. Además, el sistema proporciona un repositorio de modelos de entidad que contiene los modelos reales y generados para fomentar el intercambio de conocimientos entre los investigadores en el campo [MENDONCA BRANCO COWAN 2009].

S.P.L.O.T. es un sistema basado en la Web y desarrollado en Java 2 que utiliza un motor de plantillas HTML para construir interfaces de usuario altamente interactivas para configuración y razonamiento basadas en Ajax. Considerando que el sistema está basado en Internet, esto facilita el intercambio de conocimientos (por ejemplo, el repositorio de modelo de características), sin necesidad de descargar las actualizaciones del software [MENDONCA BRANCO COWAN 2009].

Link: http://www.splot-research.org/

3.2. Productos Comerciales Relacionados

3.2.1. MYGENERATION

CODE GENERATION, O/R MAPPING, AND ARCHITECTURES

Es una herramienta para la generación de código basado en plantillas (*templates*). Las funciones del producto permiten editar cada estilo de programación en un editor integrado. La generación trabaja estrechamente con una conexión a la base de datos, de la base de datos, se extraen las características sobre cada tabla a fin de generar el código.

La versión de la herramienta que se analizó, sólo considera la generación de código *Back-End*, lo cual la limita en el paradigma de línea de producto de software. A pesar de eso, utiliza parte de los métodos de ingeniería de dominio y programación generativa.

La herramienta, tampoco considera la definición de un conjunto de componentes al mismo tiempo, que entre otras cosas, no permite la implementación de un *framework* que requeriría la combinación de varios componentes de software.

Link: http://www.mygenerationsoftware.com/portal/default.aspx

3.2.2. Code Generation and T4 Text Templates

Code Generation es solo una utilería o *plug-in* para la generación de archivos de texto. Está integrada en el *IDE (Integrated Development Environment)* de Visual Studio 2012. El intérprete considera un conjunto de propiedades para manipularlas y generar el texto (*o código*), las propiedades pueden estar integradas en los recursos o en archivos externos.

Este *plug-in*, es solo un ejemplo de programación generativa limitada. El diseño del lenguaje, que pretende generalizar su uso a través de la manipulación de propiedades, también es limitado. Además, de que la incorporación al Visual Studio limita su utilización a un contexto tecnológico que no está dedicado a la generación de código.

Link: http://msdn.microsoft.com/en-us/library/bb126445.aspx

3.2.3. Template-Based Code Generation

Se trata de un conjunto de bibliotecas que utilizan un enfoque similar, de la forma como trabaja el diseño del intérprete de esta investigación. En este producto, el modelo de datos es integrado en una especificación *XML*, que después es manipulado por un *parser* llamado *SAX*. El *parser* se encarga de proporcionar los valores a clases y métodos que se dedican a generar el código. La implementación de los *templates*, se hace en lenguaje Java y el código generado, también es exclusivamente Java, ver Figura 14.

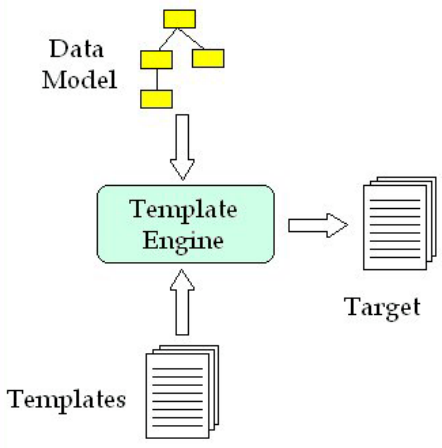

Figura 14. Generación de código basado en templates.

Estas librerías son creadas como *open source*, y su implementación sólo intenta que algún experto en las tecnologías Java automatice su trabajo. Lo anterior se debe a la dificultad en torno al uso de las funciones de las clases y su limitación en el producto final.

Link: http://onjava.com/pub/a/onjava/2004/05/05/cg-vel1.html

3.2.4. BIGLEVER SOFTWARE GEARS SPL LIFECYCLE FRAMEWORK

Gears Product Line Engineering Tool y Lifecycle Framework, permite diseñar un portafolio de Línea de Productos completo como un solo sistema de producción más que como un sistema de múltiples productos. Con esta innovadora capacidad de "sistema único", se puede desarrollar y gestionar la línea de productos sin problemas y eficientemente, a través de cada etapa del ciclo de vida – desde los requerimientos hasta el diseño, implementación, pruebas, entrega, mantenimiento y evolución.

El marco de trabajo *Gears* provee un conjunto de estándares industriales *PLE* (*Product Line Engineering*) con conceptos y pre-construcciones que dan valor agregados a sus herramientas, activos, y procesos en todo el ciclo de vida. Entre sus principales características están las siguientes:

- Un modelo de características (*feature*), que se utiliza para expresar la diversidad de propiedades (opcionales y diferentes características a elegir) entre los productos en la línea de producción.

- Un mecanismo de punto de variación uniforme que está disponible directamente dentro de las herramientas y sus componentes asociados. Puede ser usado para el manejo de la variación por medio de características en toda la etapa del ciclo de vida de ingeniería.
- Un configurador de producto que se usa para ensamblar automáticamente y configurar los componentes y los puntos de variación – basado en la selección de características que se hace en el modelo *feature* – produciendo todos los componentes para cada producto en la línea de producción con solo presionar un botón.

Link: http://www.biglever.com/

Características citadas por: [KRUEGER CLEMENTS 2013]

CAPÍTULO III

DISEÑO DE LA PLATAFORMA DE SOFTWARE

1. INTRODUCCIÓN

En este capítulo explicamos la propuesta de diseño que se implementa en una plataforma de línea de productos de software. La plataforma es una herramienta con un diseño tipo *RAD (Rapid Application Development)* y su objetivo es generar aplicaciones informáticas utilizando el paradigmas de línea de productos de software con técnicas de programación generativa.

Comenzaremos describiendo el método de generación propuesto, para después abordar el diseño de la herramienta, haciendo especial énfasis en la configuración de la arquitectura y sintaxis de lenguajes, que resuelven las problemáticas iniciales de la investigación.

Finalmente, se explican las partes medulares de la solución, es decir, el diseño de la variabilidad de la arquitectura y el lenguaje de dominio específico.

2. ENFOQUE DE SOLUCIÓN

La herramienta propuesta debe considerar la producción de más de un sistema a la vez, por lo tanto, una de las premisas es contemplar los requisitos necesarios para gestionar varios proyectos. Como parte de nuestra herramienta de gestión, se incluye una capa de persistencia de datos, y en este caso, los datos son características relevantes de cada línea de producción.

Como parte del diseño, también se ha considerado la posibilidad de generar sistemas usando diferentes arquitecturas y lenguajes de programación. Cada una de las arquitecturas, sus componentes y sintaxis, son incluidos, en un archivo de configuración reservado para la línea de producción.

El interés de integrar la configuración de arquitectura, componentes y sintaxis de modo independiente, tiene como objetivo aumentar la flexibilidad de la herramienta para permitir modificar o agregar nuevas capacidades, de tal suerte que sea posible incluir una nueva línea de producción con su arquitectura y componentes correspondientes. Los componentes son un conjunto de programas y archivos que forman parte de la arquitectura y que tienen un comportamiento u objetivo específico.

La arquitectura de la línea de producción se configura sobre una solución diseñada para soportar las distintas variantes. La administración de la variabilidad se logra usando un modelo de características (*features*) que permite considerar diferentes condiciones y

componentes de la arquitectura. En la implementación de la solución, se ha decidido usar un modelo de variabilidad jerárquica para líneas de producción de software.

Para lograr la independencia en la generación de código de los programas, se diseñó un lenguaje intermedio, donde se expresan las distintas sintaxis. El proceso de generación considera como insumos a las características relevantes de los sistemas y al código del lenguaje intermedio. Las características relevantes son capturadas por el usuario de la herramienta y el código intermedio es integrado por los arquitectos y programadores que diseñan y configuran cada línea de producción. Como resultado final tenemos un código para cada componente con las características requeridas.

Una buena generación de código, aproxima un sistema hacia un producto de calidad, e incluye un código libre de errores, estandarizado y debidamente documentado. Al momento de editar los componentes en el lenguaje intermedio, se tiene que enfatizar la codificación de todas las características que se quieren tener en el código final; y es ahí, donde se preparan los estándares en nomenclatura, interfaz de objetos y utilización de bibliotecas para entidades externas.

Con la aproximación de los sistemas, se logra un entorno de software previamente armado con todos los componentes generados y necesarios. Este entorno tiene las características previamente diseñadas por un equipo de expertos en arquitectura, estándares y mejores prácticas en la codificación; por lo cual se asegura, que la calidad del producto final es buena.

Documentación

La iniciativa de automatizar la documentación de los sistemas obedece a la motivación principal de tener un producto de calidad. Del mismo modo en que se genera el código de cada componente, se automatiza la generación de su respectiva documentación. Los estilos de documentación forman parte de la configuración y pueden ser modificados o agregados como parte del archivo de configuración. La integración de los estilos de documentación a la configuración, soporta adaptaciones a los distintos requerimientos particulares de imagen y formato.

El último punto que se debe considerar como parte de la documentación, es el proceso de recuperación de comentarios desde los componentes codificados. Para esto se debe considerar una configuración basada en expresiones regulares para rescatar los comentarios. El proceso de rescate de los comentarios revisa cada componente de nuestra aplicación y rescata los segmentos de explicación relevantes.

El proceso de recuperación de comentarios se hace preferentemente al finalizar la etapa de construcción, esto con el fin de contar con la última versión de los componentes y sus comentarios. Cabe hacer notar, que la etapa inicial de aproximación, fue completamente superada por las afinaciones de cada componente.

En resumen, la documentación final, tendrá muchos comentarios de la aproximación inicial, así como otros comentarios del proceso de afinación en la construcción.

Actualmente, existe la práctica de recuperar manualmente los comentarios desde el código, y esta recuperación es realizada por parte del equipo, lo cual genera costos altos y gran parte del tiempo, la sub utilización de recursos calificados. Esta práctica siempre es necesaria y suena muy lógica, la propuesta es que sea automatizada por una herramienta.

3. MÉTODO DE GENERACIÓN DE SISTEMAS

Se propone un método de construcción de un proyecto de software, basándose en la operación de la herramienta y en los insumos que son requeridos para sacar la mayor ventaja en la construcción de un sistema. Este método puede ser aplicado parcialmente si así se desea, dejando la generación de código *Front-End* fuera de la misma.

Los pasos propuestos de la Figura 15, sólo son para la etapa de construcción de proyectos de software y no consideran la configuración de la línea de producción en la herramienta.

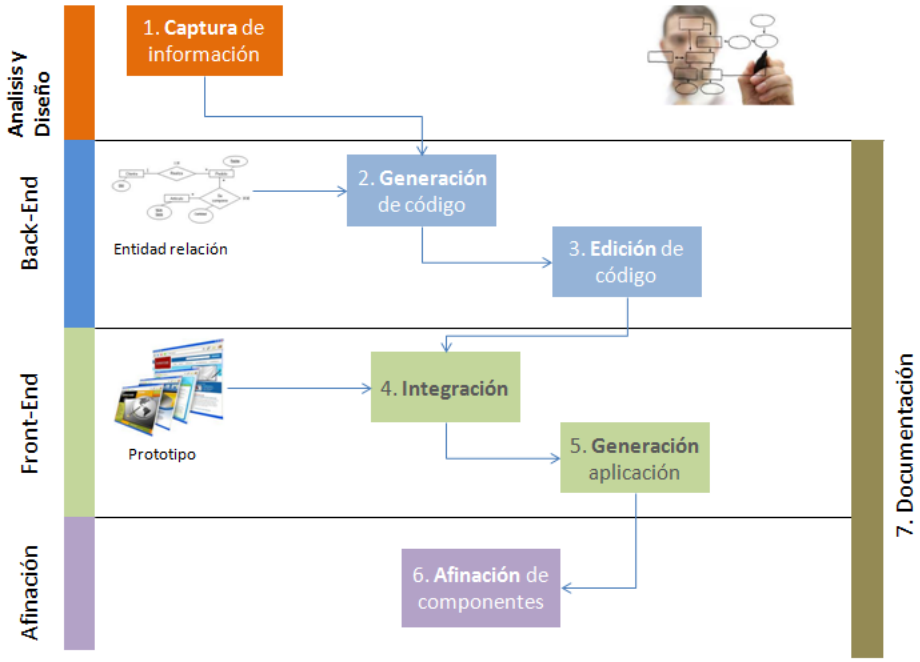

Figura 15. Método de generación de sistemas.

Actividad 1. Tiene que ver con la captura de la información que ha resultado del análisis y diseño de un sistema de software. Ésta es información general, la cual después es recolectada para la automatización de la documentación.

Actividad 2. Es la generación del código *Back-End*. Aquí es necesario integrar el modelo entidad-relación del sistema, documentarlo y generar el código seleccionando alguna línea de producción.

Actividad 3. La edición de código es responsabilidad de los programadores de la lógica de negocio. La edición tiene que considerar seguir con los mismos estándares que se aplican durante la generación. Además es importante exigir que el código se documente a semejanza de lo que se generó.

Actividad 4. La integración del prototipo y los métodos de negocio, requiere dos insumos principalmente: el prototipo de la aplicación y el código *Back-End*. En esta actividad, la herramienta analiza el código del prototipo y del *Back-End* y los presenta al usuario para su configuración.

Actividad 5. La generación de la aplicación, esta es la última participación de la herramienta en la generación de código. La herramienta se encarga de recolectar toda la información disponible, y con ayuda de la línea de producción, el código del sistema.

Actividad 6. La afinación de la aplicación le deja toda la responsabilidad a los programadores para probar y terminar el código generado por la herramienta.

7. Documentación. La automatización de la documentación puede ser considerada desde el primer momento que se tiene información sobre el sistema.

4. DISEÑO DE LA HERRAMIENTA

El diseño de la herramienta es igual a un sistema de administración de aplicaciones, la cual debería estar construida en un lenguaje de programación que incluya librerías para conectarse a una base de datos, manejar archivos empaquetados .zip, utilizar expresiones regulares y generar documentación.

Finalmente, tenemos una herramienta especializada pero con características iguales que cualquier aplicación informática, donde se contará con una interfaz de usuario para administrar los insumos y unos procesos para generar los productos correspondientes, ver Figura 16.

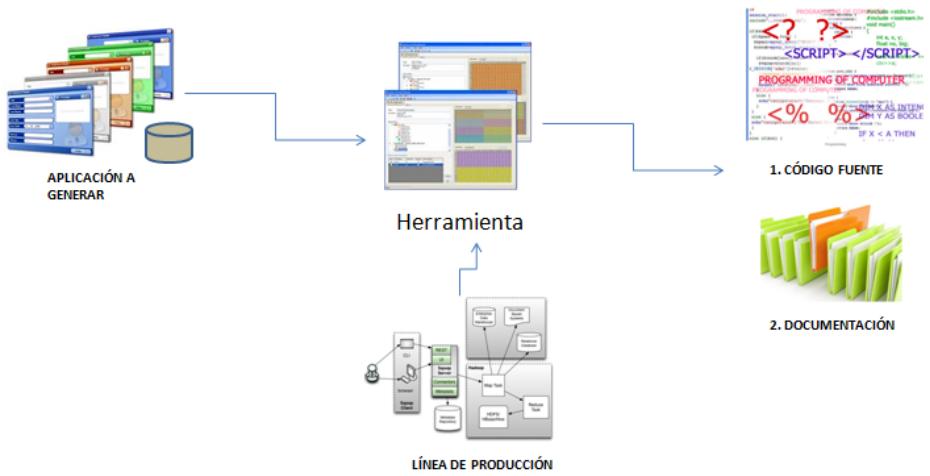

Figura 16. La Herramienta.

La información relativa a cada uno de los sistemas que entran a la línea de producción, es registrada en una base de datos. Estos datos nos permiten aportar la información necesaria para generar los productos sobre los mismos sistemas. La integración de los datos referentes a los detalles técnicos de cada sistema, se logra a través de la interfaz del usuario. Esta captura nos permite concentrar la información relevante en un solo contenedor, para que sea utilizada y manipulada a la hora de generar los productos.

La definición de cada línea de producción tiene lugar en archivos con formato *XML* disponibles como parte de la configuración. La configuración es tomada en cuenta por la herramienta al momento de generar los productos de salida por cada sistema. La configuración de la arquitectura también comprende archivos que utilizan el lenguaje intermedio para definir los componentes de software que se generan.

Los procesos de generación de código y automatización de documentación consideran la información relevante de cada sistema y la manipulan utilizando un proceso de interpretación del meta-lenguaje y la definición en plantillas (*templates*) de documentación.

La generación de cada componente se hace a través de un intérprete que acepta como insumos los componentes de la arquitectura y las características de los sistemas. Cada componente de la arquitectura configurada, está segmentado en una parte fija y una parte variable. La parte fija de un componente tiene la sintaxis del lenguaje seleccionado, y la parte variable se encuentra escrita en el lenguaje intermedio llamado *LENCOS* (ver Figura 17).

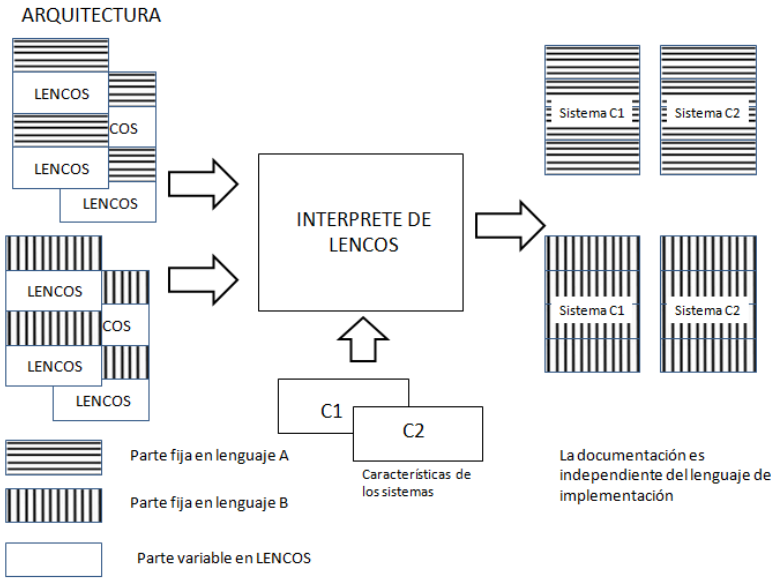

Figura 17. Intérprete de LENCOS.

El intérprete de *LENCOS*, escribe cada componente copiando la parte fija y resolviendo la parte variable con ayuda de las características del sistema. El conjunto de todos los componentes interpretados, representan el producto de software de la línea de producción.

4.1. Insumos

Los insumos de la herramienta son de origen distinto, se tienen pantallas de captura de datos, carga y procesamiento de archivos. Las pantallas de captura de datos son la interfaz con el usuario para complementar toda la información requerida de los sistemas. Esta información incluye el nombre de la aplicación, responsables, entidades de datos, descripciones, campos, etc. Los datos relativos a una aplicación son utilizados tanto para generar el código como para documentar.

A modo de automatizar la carga de información, los archivos de insumos pueden ser definiciones de base de datos en lenguaje de definición de datos *DDL*. Estos archivos son procesados por un procedimiento de la herramienta (y una configuración de expresiones regulares. *Ver apéndice B*). También es posible cargar archivos con formato *HTML* para el prototipo e imágenes para la documentación de pantallas.

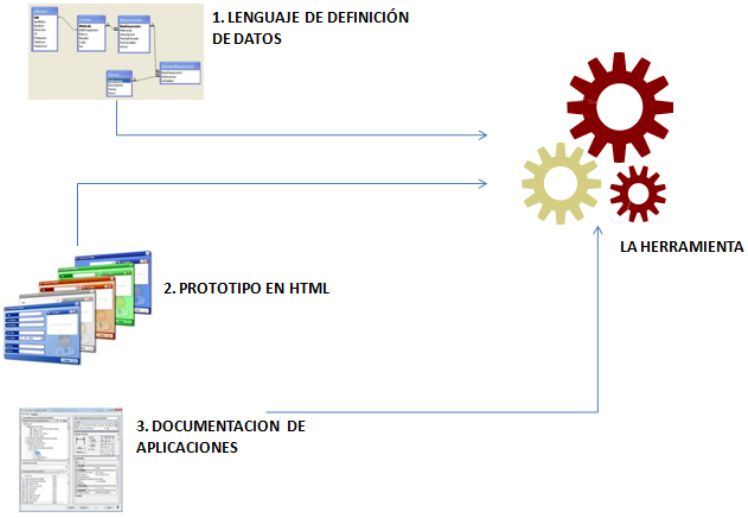

Figura 18. Los Insumos.

El lenguaje de definición de datos (*DDL*) del punto 1 en la Figura 18, es un *script* estándar, que define cómo se encuentran dispuestas nuestras entidades de datos y cuáles son los campos que la integran. Esta definición es procesada por la herramienta para ahorrar tiempo en la captura manual de esta información. La captura de datos de forma precisa y correcta para cada entidad, tiene implicaciones directas sobre el código generado y los comentarios; además de que los mismos datos son considerados en los procesos de automatización de documentación. Ejemplo de lo anterior es el diccionario de datos.

La información de las entidades y sus campo tiene una relación directa con la generación del código *Back-End,* cuya responsabilidad es la manipulación de la persistencia de datos. En el caso de estudio que nosotros utilizamos, se genera una clase por cada entidad de datos y una variable por cada campo, así mismo, por cada variable se generan sus interfaces públicas *set* y *get*. Además de esto, se genera el código respectivo para el acceso a la capa de persistencia y el código para la capa de negocio.

El prototipo en *HTML* del punto 2 en la Figura 18, se trata de un *script* definido previamente por los analistas y diseñadores gráficos, pero validado por los usuarios. Cada archivo participa al momento de hacer la integración entre el *Front-End* y su respectivo código *Back-End*. El proceso de integración se encarga del mapeo de las pantallas con las clases e interfaces de los métodos de negocio del *Back-End*.

La documentación de cada sistema del punto 3 en la Figura 18, es información relativa a cada sistema que se desea aproximar, sus pantallas, sus reportes, sus fuentes de datos, etc. Esta información es utilizada tanto en la generación de código, como en la automatización de documentación. El objetivo de una interfaz de usuario para la captura desde la herramienta, es disminuir costos en la documentación de cada sistema, ya que se ha notado que esta actividad suele ser redundante.

Como vimos anteriormente, la persistencia de datos de la herramienta está dispuesta en dos partes. Por un lado tenemos una base de datos normal, con sus respectivas tablas donde se guarda la información necesaria para documentar los sistemas; y por el otro lado, tenemos una biblioteca de configuración, para ampliar las capacidades de la herramienta en cuanto a las líneas de producción (*arquitecturas y estilos de documentación soportados*).

4.2. Productos

El producto principal de la herramienta es código fuente generado, que forma parte del ciclo de desarrollo de software. Sin embargo, tenemos que considerar que persiste información de cada sistema en la herramienta y que puede ser manipulada para lograr nuevos productos. Hasta ahora, el diseño de la herramienta ha considerado ejemplificar la automatización de documentación para un diccionario de datos, un diseño funcional y un diseño técnico.

En el proceso de generación de código fuente, es necesario tomar en cuenta que la herramienta se asegura de que el diseño del nuevo sistema utilice el modelo vista controlador (*MVC*). Por lo tanto, se consideran al menos tres capas: *Front-End, Back-End* y persistencia de datos. La capa *Back-End* contiene los *beans*, lógica de negocio y el componente de acceso a datos. La capa *Front-End* contiene la interfaz del usuario, validaciones de datos y las llamadas a la interfaz de métodos de negocio. La capa de persistencia resguarda los datos de cada sistema, ver Figura 19.

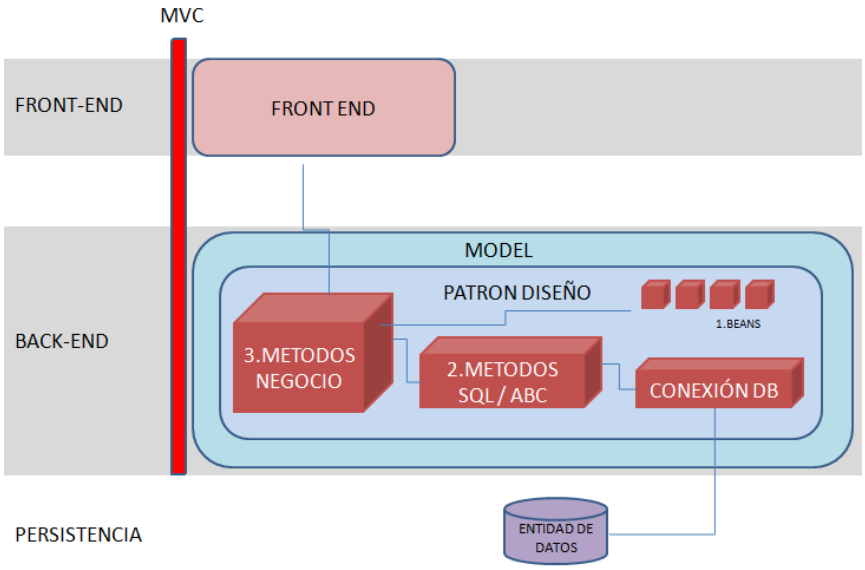

Figura 19. Modelo Vista Controlador.

La información es capturada a detalle para la capa del modelo vista controlador. En la capa de persistencia tenemos documentada cada entidad, campos, tipos de datos y llaves primarias. En la capa de *Back-End* tenemos un registro de las interfaces de negocio que son utilizadas. Y por último, en la capa *Front-End* tenemos documentada cada pantalla, campo, validación de datos y la relación entre las pantallas que el usuario visualizará en el producto final.

La definición de las entidades y sus campos son la base para generar código *Back-End*. Como ejemplo, en el patrón de diseño *J2EE* por cada entidad y sus campos se genera una clase con propiedades. Adicionalmente se genera una clase específica para los accesos a la capa de persistencia y otra clase que incluye la lógica de negocio y el armado de los mecanismos de comunicación entre objetos.

La primera generación de código llevada a cabo es la generación del *Back-End* y posteriormente se integra al proceso de generación del *Front-End*. Sin embargo, la generación del código *Back-End* puede ser manejada de manera independiente e incluso, ser utilizada sin integrarse con la parte del *Front-End*. Esta parte se ha pensado

así para no tener restricciones en el uso de la herramienta, por ejemplo, si no tenemos definido el estilo de programación para el *Front-End*, entonces podríamos generar solamente el código del *Back-End* y desde ahí se programan de forma artesanal las pantallas de interfaz con el usuario.

4.3. Meta-arquitectura (Variabilidad)

Se propone una herramienta para la generación de código en varios lenguajes de programación, y para ello se diseña un lenguaje de dominio específico que asegura que se cumpla ese objetivo. Sin embargo, el diseño también incluye la especificación de una arquitectura en la organización de los componentes de software.

Para lograr la implementación de arquitecturas distintas, se requiere hacer una definición usando una combinación de archivos de configuración y componentes. De este modo, logramos por medio de la herramienta hacer implementaciones de distintos *patrones* y *frameworks* como los utilizados por Java (*Struts*, *Spring*), así como también para generar aplicaciones usando ASP o ASP.NET 4.0. Un ejemplo de lo que es un archivo de configuración, lo podemos encontrar en el *Apéndice C*.

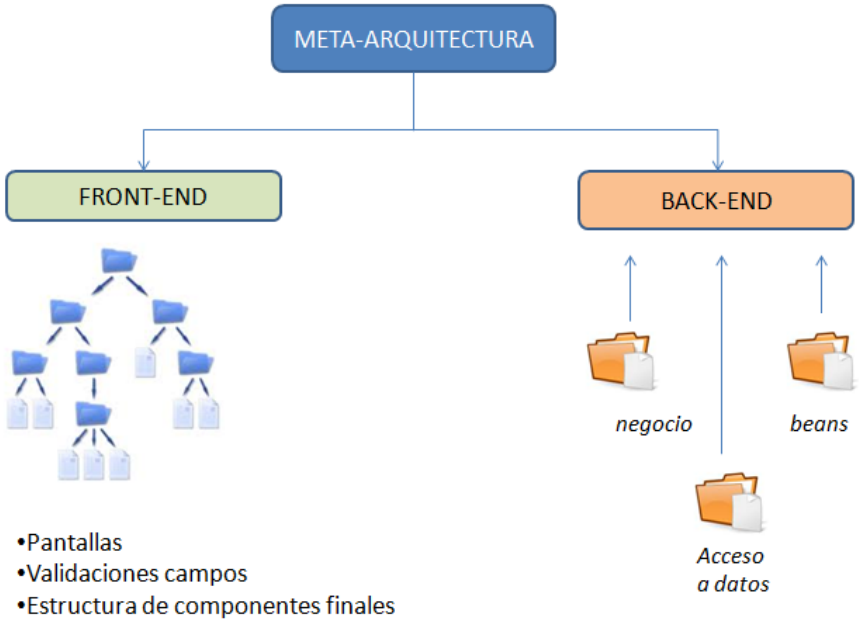

Figura 20. Meta-Arquitectura.

El diseño del repositorio permite también tener especificaciones de diferentes arquitecturas, paradigmas y estilos de programación (Figura 20). Para lograr distinguir una configuración de otra, se organizan los archivos de configuración en directorios con nombres distintos. Físicamente, estos directorios se encuentran en un archivo con formato *Zip*, lo que nos permite incluir nuevas configuraciones y distribuirlas sin necesidad de actualizar el código de la herramienta.

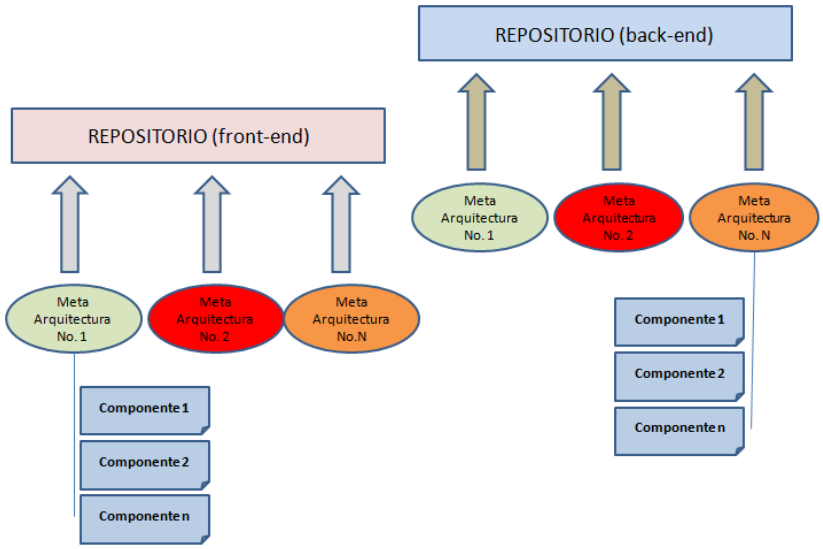

Figura 21. El Repositorio.

En el momento que son procesados los productos de la línea de producción, cada proceso entra al archivo de configuración, descompacta los componentes que necesita y los procesa en su formato original (Figura 21). El proceso de des-compactación automático sucede tanto en la generación de código, como en la automatización de la documentación donde la herramienta toma múltiples archivos en formato Word. Los procesos que hacen uso de la configuración, extraen cada archivo en un directorio temporal, los procesa y finalmente los elimina. En cualquier proceso de la herramienta, nunca se modifican los componentes originales.

4.4. Meta-lenguaje

El Lenguaje intermedio que llamamos *LENCOS* (*Lenguaje para la Conversión de Servicios*), fue diseñado con el fin de ser lo suficiente flexible para generar cualquier código a partir de dos insumos. Un contexto de valores dispuestos en un servicio *XML* y un *script* que resalta la sintaxis requerida en el código resultante.

Considerando lo general que puede llegar a ser *LENCOS*, lo único que resta decir, es que si se requiere adaptar el lenguaje a otra herramienta, basta con hacer una implementación que forme al vuelo un servicio *XML* para posteriormente someterlo al intérprete de *LENCOS*.

La siguiente Figura es un esquema general del lenguaje y su estructura de ejecución:

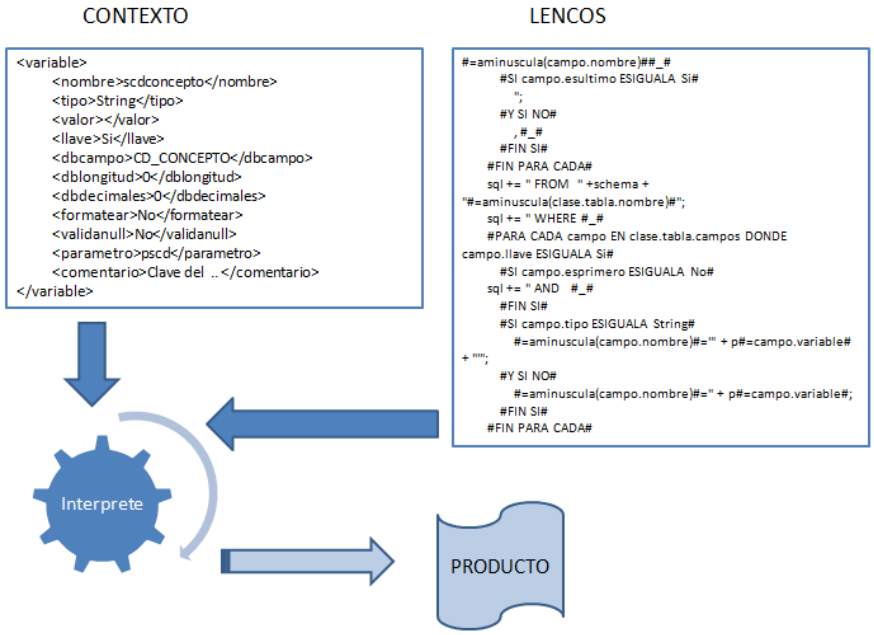

Figura 22. La Ejecución de LENCOS.

En la Figura 22, después de que cualquier aplicación o módulo arma el *XML,* éste pasa a formar parte, de la alimentación del intérprete de *LENCOS*. Además, el intérprete se alimenta del *script* con la programación de la sintaxis que se quiere en el código de salida. Al ser ejecutado el *script*, se manipula la estructura entregada en el contexto de valores, y se logra el resultado ejecutando cada línea del *script* de *LENCOS*.

4.5. Tecnologías de implementación

Una vez terminado el diseño de la herramienta, se seleccionó el *Visual Basic .NET* como el lenguaje de programación para la implementación porque tiene un tratamiento nativo a los productos de *Microsoft Office* y tiene librerías para el manejo de expresiones regulares. La definición del manejador de base de datos es un punto sencillo, considerando que la mayoría cumple con los requerimientos necesarios ya que se decidió manejar la base de datos solo en su característica de contenedor de datos, sin

hacer uso de funciones o procedimientos particulares a cada producto. Para el este caso particular se utilizó SQL Lite, ver Figura 23.

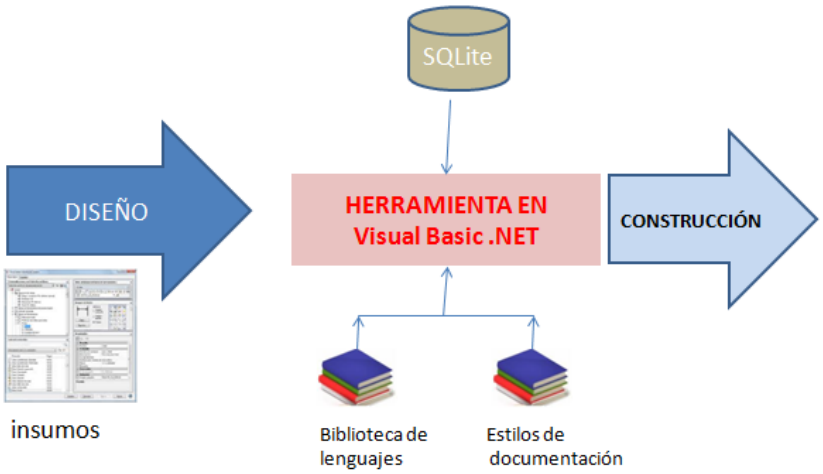

Figura 23. Tecnologías de implementación.

Como mencionamos anteriormente, la configuración de cada línea de producción se encuentra definido en archivos *XML*. Cada línea está separada en repositorios y los archivos que lo integran tienen el objetivo de generar los componentes requeridos para un sistema. Cada vez que se tiene una nueva versión de la configuración se empaqueta en un archivo *Zip.*

El modelo entidad-relación de la base de datos considera registrar la información de las pantallas y la información de las entidades de datos que participan en un proyecto. A partir de la manipulación de esta información, el sistema forma un *stream* de datos en *XML* para la generación de código y automatización de documentación.

4.6. Diseño de persistencia de datos

La herramienta tiene esquema de persistencia de datos compuesto por la información general de los sistemas y la configuración de líneas de producción.

Como habíamos visto anteriormente, en la base de datos de la herramienta se tiene registrada la información de cada aplicación como es la información general, características de la interfaz de usuario y de sus entidades de datos.

La información es modelada tomando en cuenta cada sistema como parte central del modelo, de ahí se integra la información relativa a las pantallas del *Front-End,* el detalle de los campos relevantes y validaciones de los datos del usuario.

4.6.1. El modelo Entidad-Relación del Front-End

El modelado entidad-relación para la base de datos que utiliza el *Front-End,* tiene el objetivo de ser usado para la información de cada sistema, tanto para la generación del código como para la automatización de la documentación. Entre las entidades de datos principales están aquéllas que resguardan la información de las pantallas y sus campos de captura y los reportes. Para las aplicaciones web también se incluyen los eventos de navegación.

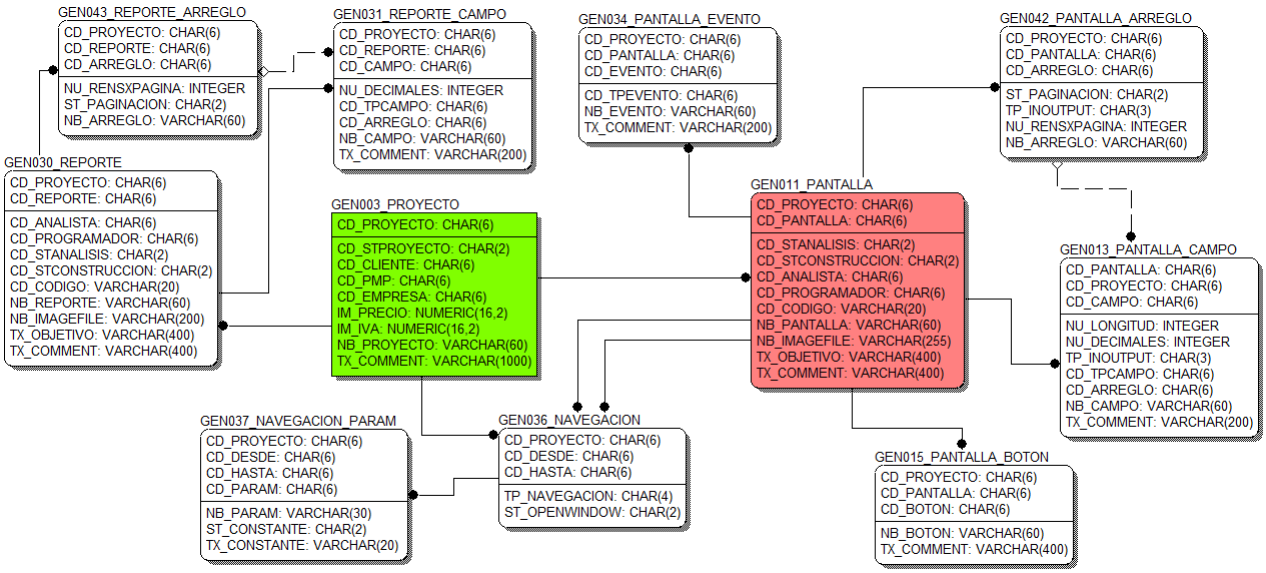

Figura 24. Modelo *entidad-relación* del Front-End.

En la Figura 24 podemos darnos cuenta que nuestro modelo atiende el contexto de los reportes y el contexto de las pantallas.

El contexto de los reportes se limita a documentar los campos que participan en el informe, así como los arreglos de los mismos. Estos arreglos son las agrupaciones de los campos, es decir, grupos de campos tienen por objetivo repetir los renglones al momento de imprimir sus datos.

En el contexto de las pantallas se consideran los botones, campos, arreglos, eventos y para los sistemas web, algunos eventos de navegación.

La documentación del *Front-End* tiene dos niveles de documentación. El primer nivel es de información general y el segundo es la información específica de funcionalidad. Estos dos niveles de información son cargados en momentos diferentes, el primer nivel de información se ingresa desde el principio que se tiene conocimiento de cada pantalla y el segundo nivel al momento de integrar el prototipo en formato *HTML*.

El nivel específico de funcionalidad, requiere un conocimiento a detalle de la implementación en las pantallas, porque ahí, prácticamente se está definiendo la inter-relación que existe entre los métodos de negocio y la interfaz del usuario. Una vez que se termina de definir el nivel específico de la funcionalidad, ya se está listo para aproximar la aplicación.

Es importante recordar que el proceso de aproximación está dividido en dos etapas: La generación de código *Back-End* y la generación de código *Front-End*. La etapa de generación del *Back-End* puede ser tratada de manera independiente e incluso, pueden ser utilizados sus resultados sin necesidad de pasar a la integración con el *Front-End*.

4.6.2. El modelo Entidad-Relación del Back-End

El modelo entidad-relación del *Back-End* registra la información de las entidades de datos y sus campos. Varias entidades de datos se integran en una fuente de datos, y estas fuentes de datos se relacionan con un sistema. Las fuentes de datos normalmente son bases de datos y las entidades son tablas, procedimientos almacenados o vistas; también es válido que la fuente de datos sea un conjunto de servicios con arquitectura *SOA (Service-Oriented Architecture)*, donde cada entidad representaría a un servicio en particular.

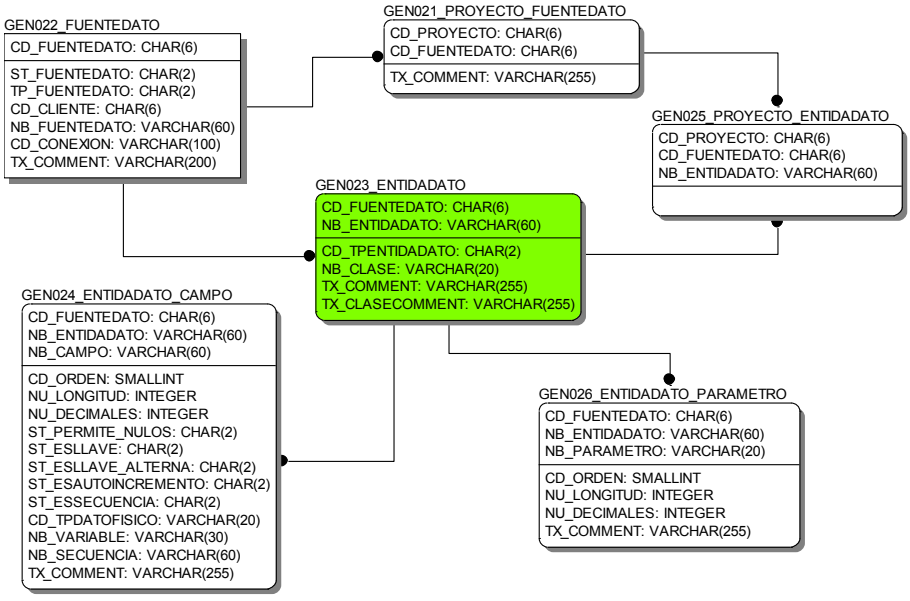

Figura 25. Modelo entidad-relación del Back-End.

La Figura 25 muestra el modelo entidad-relación donde se representan las entidades para la capa de persistencia de cada sistema, y es ahí donde se busca la información relativa a las entidades y sus campos. Para las veces que la entidad de datos represente un procedimiento o función, también es posible tener un registro de los parámetros.

Los procedimientos de la herramienta, se encargan de recuperar la información de la fuente de datos, para después representarla en *XML.* Esta representación posteriormente forma parte de los insumos del proceso de generación de código y automatización de la documentación.

En el caso del estudio que se documenta en el capítulo IV, la línea de producción utilizada está diseñada de tal modo que cada entidad de datos sea representada en un archivo con la declaración de una clase; y que los campos que forman estas entidades sean representados por variables dentro de la clase. La representación en código depende solamente del estilo de codificación que se selecciona en el proceso de generación de código.

4.6.3. El contenedor de la configuración

La configuración contiene todo aquello que aumenta la capacidad de la herramienta y que puede ser actualizada con solo copiarla por separado, sin necesidad de actualizar la versión de la herramienta. Particularmente se ha diseñado así, para poder incluir otras líneas de producción, componentes y estilo de documentación.

La configuración está organizada jerárquicamente en directorios. En el directorio raíz hay un archivo pivote que se encarga de direccionar las peticiones hacia los subdirectorios; cada uno de los subdirectorios principales forman una línea de producción.

El proceso de consulta de la línea de producción regresa el conjunto de componentes para resolver peticiones de generación de código o automatización de documentación. Cada subdirectorio incluye un archivo de configuración para administrar la línea de producción. El formato de cada archivo depende de su objetivo, por ejemplo *XML*, meta-lenguaje, Word, Excel, imágenes, etc.

En el pivote de acceso principal tenemos las referencias a las líneas de producción. Cada línea de producción inicia con un archivo de configuración interno (punto de acceso) que mapea todos los componentes que son necesarios.

Las líneas de producción, tienen referencia a archivos en lenguaje intermedio *LENCOS*, que son componentes para completar el entorno de desarrollo, además, en el punto de acceso, se tiene referencia a parámetros necesarios para la generación. Estos puntos de acceso son abordados en el momento en que se procede con la generación de código, tomando en cuenta primero los parámetros, y procesando después los programas en *LENCOS*; al final de la generación, se integra la lista de componentes faltantes para completar el entorno de desarrollo.

Los puntos de acceso para las líneas de producción de documentación, hacen referencia a un archivo principal en formato Word, una lista de macros y una lista de variables. Este punto de acceso se procesa al automatizar la documentación del sistema. La automatización de la documentación toma en cuenta el archivo principal para iniciar el formateo, en el archivo principal se encuentran referencias a las macros que a su vez integran otros archivos Word; los valores de las variables que se requieren al usuario, son sustituidos en cada referencia dentro del documento.

4.7. Procesos e interfaz de usuario

Las interfaces de la herramienta se han diseñado tomando los criterios de orden y sencillez en la medida de lo posible. Cuando la dificultad del proceso es complicada, se diseña un flujo de pantallas paso a paso a modo de *wizard*.

Considerando que actualmente la herramienta tiene muchas funciones, aquí solo vamos a describir los procesos que son identificados en la Figura 26.

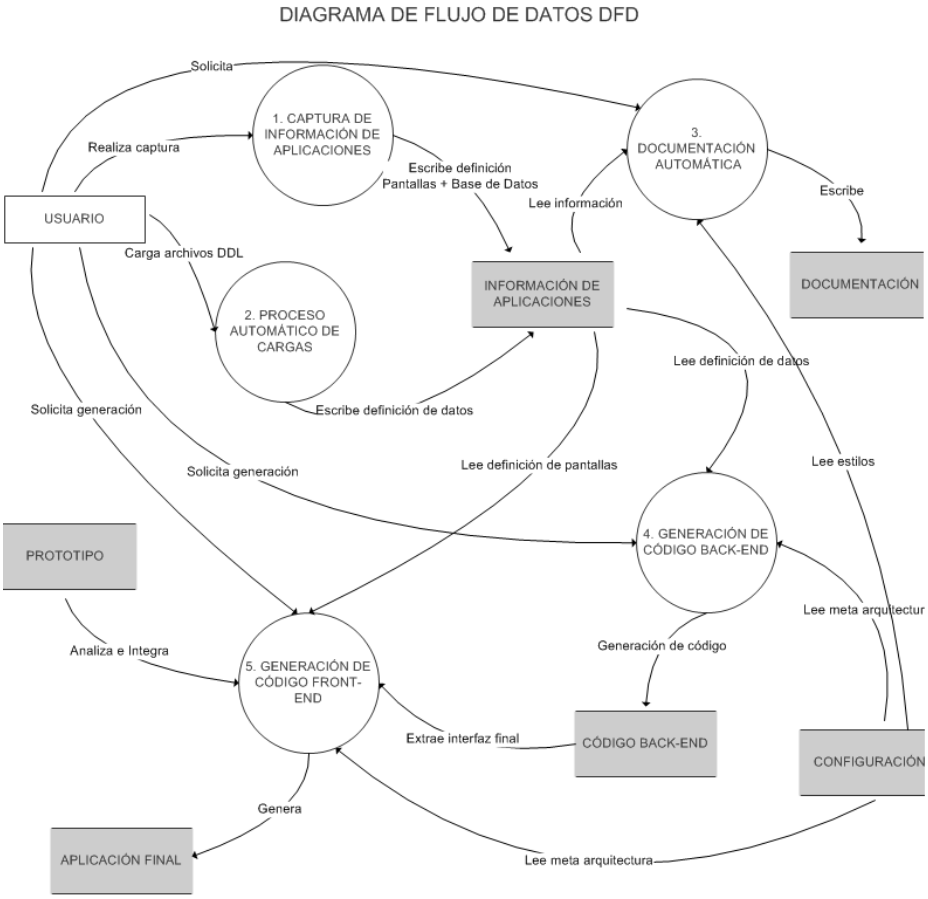

Figura 26. Diagrama de flujo de datos general.

En la Figura 26 debemos destacar que la única entidad externa es el usuario es responsable de integrar los insumos de la herramienta y ejecutar los procesos de acuerdo a la metodología. Las funciones para integrar los insumos son: la información de la aplicación (*proceso 1*) y ejecución del proceso de cargas automáticas de archivos de definición de datos (*proceso 2*).

El usuario también es el responsable de solicitar los procesos para la generación de los productos como son: automatizar la documentación (*proceso 3*), generación del código *Back-End* (*proceso 4*) y por último, el proceso de integración *Front-End* (*proceso 5*).

Las únicas unidades de almacenamiento propias de la herramienta son la base de datos con la información de los sistemas y la configuración. Entre los productos de salida que se escriben tenemos la documentación de cada aplicación, el código *Back-End* y el código *Front-End*.

Como se ha mencionado anteriormente, el *proceso 5* de generación de código *Front-End* es un proceso opcional. Se ha identificado que las situaciones en que no se utiliza es cuando no se tiene especificada la configuración de la interfaz *Front-End*, cuando el proyecto tiene por objetivo solo la construcción de servicios *SOA* o cuando la tecnología de interfaz de usuario no es parte de la configuración. Por el contrario, también se identificó que los procesos de automatización de documentación y generación de código siempre son utilizados, ver Figura 27.

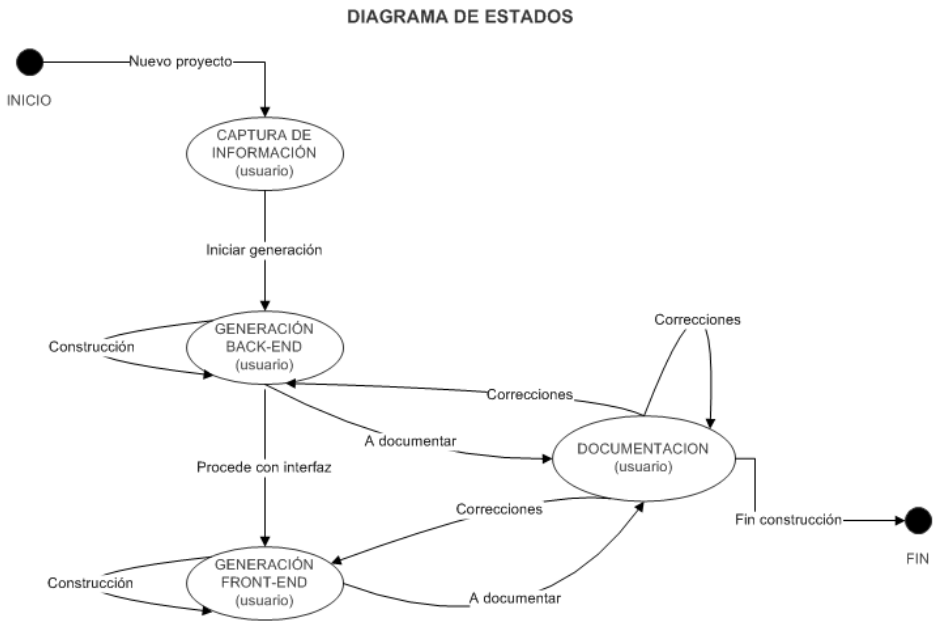

Figura 27. Diagrama de estados de una nueva aplicación.

La herramienta está construida como una aplicación de escritorio en Windows. En la construcción se usó el Visual Studio 2010 y se compila para ser compatible con procesadores de 64 bits. Para la codificación de la aplicación se ha usado Visual Basic y algunos componentes visuales en *ActiveX*. También se utilizaron bibliotecas para el manejo de expresiones regulares y manipulación de documentos en formato Word.

La herramienta se compone de una pantalla principal, la cual se divide de un menú y un área de trabajo. El usuario selecciona del menú las funciones y procedimientos que desea ejecutar y la herramienta responde con una ventana que coloca al interior del área de trabajo (*ver Apéndice D, sección 1*).

Al iniciar la ejecución de la herramienta, se conecta a la base de datos en *SQL Lite* y carga el archivo de configuración. La interacción con la base de datos y la configuración es permanente mientras se ejecute la herramienta. La única consideración adicional, es que la manipulación de la configuración requiere un directorio temporal para des-compactar los componentes necesario en los procesos.

4.7.1. Captura de información de aplicaciones

La información que tenemos por cada sistema tiene que ver con las pantallas de captura, los reportes y las fuentes de datos que serán utilizadas. Esta información se integra a través de una interfaz del usuario que se presenta en el *Apéndice D - sección 2*.

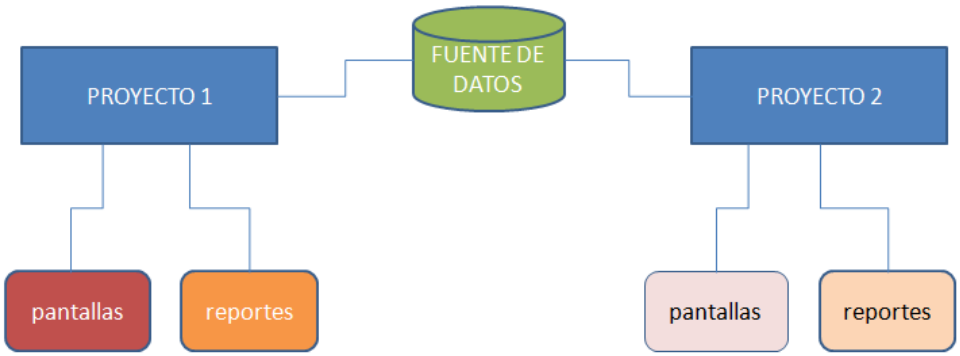

Figura 28. La relación general de entidades.

Cada proyecto tiene un subconjunto de pantallas y su información relacionada (Figura 28). Dicha información define a detalle cada campo con que se integra, sus validaciones, etc. También incluye cada reporte que se tiene y el detalle de los datos que son impresos.

Las fuentes de datos son entidades con cierta independencia que pueden ser relacionadas con más de un proyecto. Cada fuente de datos está compuesta por entidades y campos.

La información de las pantallas y las fuentes de datos son consideradas en el proceso de generación de código y la automatización de la documentación.

4.7.2. Proceso automático de cargas

El proceso de carga automática es una funcionalidad para acelerar la integración de información de las fuentes de datos. Los archivos de carga son código *scripts* en un lenguaje de definición de datos *(DDL)*. El proceso hace uso de la configuración de la herramienta para descomponer el *script* usando expresiones regulares. El *script* puede traer cualquier sintaxis y el proceso de carga la puede descomponer siempre y cuando se cuente con la configuración correspondiente. Las expresiones regulares que se deben considerar se utilizan para la identificación de entidades, la identificación de campos y la identificación de llaves primarias, ver Figura 29.

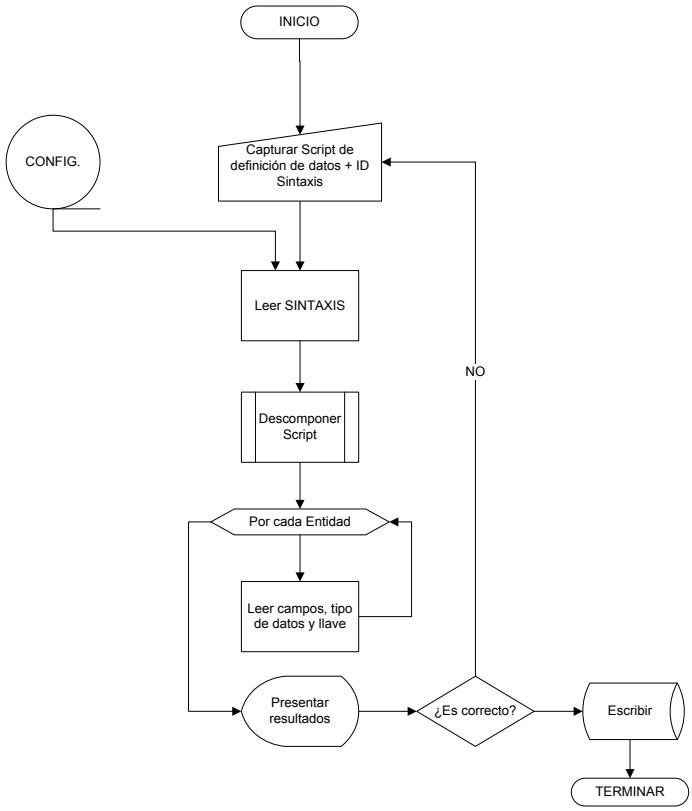

Figura 29. Algoritmo de carga automática de definición de datos.

La interfaz del usuario para esta función se ha construido a modo de pantallas paso a paso (*wizard*). El flujo de pantallas se puede observar en el *Apéndice D - sección 3*.

4.7.3. Documentación automática

El proceso de automatización de la documentación, considera como entrada la configuración, base de datos, valores de las variables y macros definidos en la configuración del formato. Cada formato o estilo de documentación, está integrado por los archivos de configuración y sus componentes en documentos Word. Los macros son requerimientos de archivos externos para ser integrados en la documentación final.

El proceso considera la información del proyecto y la integra a un servicio en un flujo *XML.* Este servicio posteriormente es manipulado para que la información sea impresa en los formatos de Word. Esta es una forma similar a la que se utiliza en el meta-lenguaje, pero se hace de forma simple (*ver Apéndice D – Sección 4 y* Figura 30).

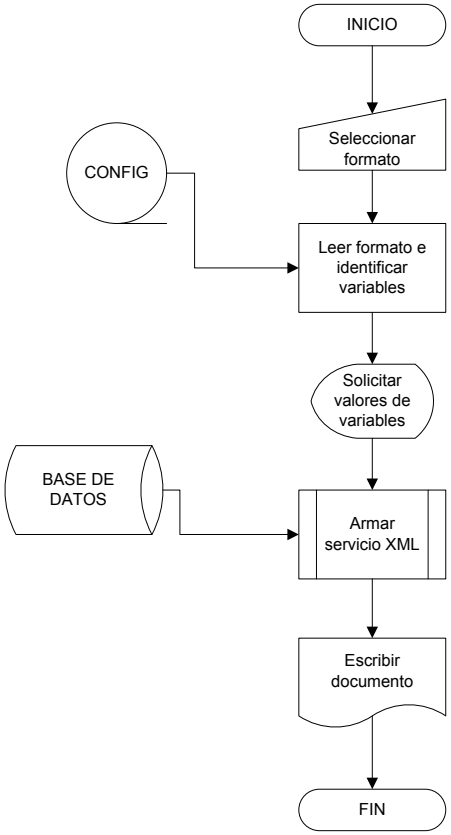

Figura 30. Algoritmo de automatización de documentación.

4.7.4. Generación de código Back-End

El proceso de generación de código *Back-End,* toma en cuenta el estilo de programación y la información registrada para las entidades. El estilo de programación se lee desde la configuración y la información de las entidades se lee desde la base de datos de la herramienta.

La interfaz del usuario presenta esta funcionalidad en pantallas paso a paso y una vez que se concluye la captura de toda la información requerida se inicia el proceso de generación de código. El diseño de las pantallas se puede ver en el *Apéndice D-sección 5.*

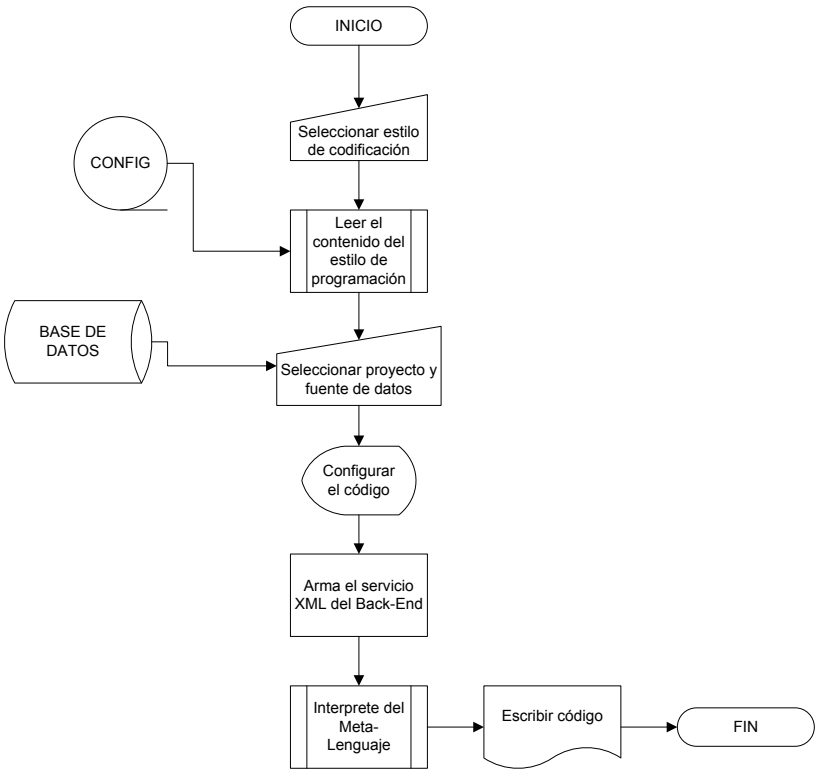

Figura 31. Algoritmo de generación de código Back-End.

La interfaz del usuario está orientada a preparar todos los insumos necesarios para el proceso de generación de código, al momento que se completan, se procede con el armado automático del servicio *XML* y con la ejecución del intérprete. El intérprete espera un servicio *XML* y el propio código en el lenguaje de dominio específico. El esquema de valores inicial se arma rescatando y manipulando la información desde la base de datos del proyecto, ver Figura 31.

4.7.5. Generación de código Front-End

El proceso de generación de código para las múltiples interfaces del usuario tiene mayor dificultad operativa y requiere ejecutar un proceso de integración por cada pantalla (Figura 32). La integración asocia la información de cada pantalla con un prototipo en *HTML*, esta asociación se hace hasta nivel de campo (ver *Apéndice D, sección 6)*.

Los prototipos deben estar preparados para ingresar a la herramienta, la preparación es necesaria para poder identificar cada uno de los campos de entrada, etiquetas de salida e identificadores de posición. Los *tags* relevantes se identifican por tener una propiedad "id", el valor de la propiedad es asociado con un campo o con una etiqueta. Los identificadores son puntos de control arbitrarios y se utilizan para insertar código.

La definición de la meta-arquitectura también tiene mayor dificultad, ya que se ha segmentado para formar cada pantalla. El producto final es un programa *CGI* (*Common Gateway Interface*) que realiza llamadas a los componentes de negocio usando las interfaces de comunicación.

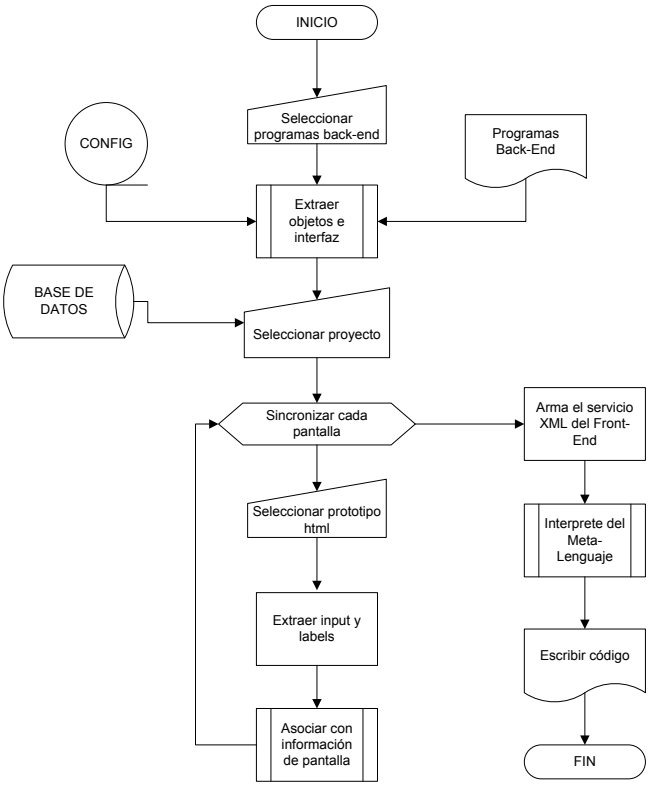

Figura 32. Algoritmo de generación de código Front-End.

Ver *Apéndice D - sección 6.*

5. LENGUAJE DE CONVERSIÓN DE SERVICIOS - LENCOS

El lenguaje se diseñó para manipular un conjunto de propiedades que afectarán un proceso de generación de texto con precisión. Para relevancia de la investigación, esta generación de texto es un *script* de código con una sintaxis.

El Lenguaje para Conversión de Servicios es un lenguaje de dominio específico y es una de las partes fundamentales de la herramienta. Es el componente mediante el cual se puede independizar la generación de código. Este lenguaje es construido como parte de la herramienta, y su objetivo es la generación de código fuente en archivos de texto.

El *script* en *LENCOS* es texto escrito en cualquier editor, tiene extensión .*met* y es ejecutado usando un intérprete específicamente construido para su sintaxis. Los *scripts* normalmente son cargados desde el interior de algún proceso para lograr la capacidad de generación de código. El resultado de su interpretación se escribe en un archivo de texto.

Todas las palabras clave de *LENCOS*, tienen que estar dentro de los caracteres delimitadores # x #, las líneas se procesan de arriba hacia abajo (*top-down*). Todos los textos fuera de los delimitadores son estrictamente respetados incluyendo los saltos de línea.

El código significativo en el intérprete, se descompone usando expresiones regulares que nos permiten identificar los valores relevantes. Cada palabra clave se asocia a un procedimiento interno para procesar los valores. La salida estándar en la interpretación siempre será dirigida hacia un flujo de caracteres, que finalmente se deja en un archivo.

El intérprete de *LENCOS* espera como entrada un espacio de valores en un formato *XML*, por lo que la interpretación de este lenguaje siempre considera las propiedades y valores que están disponibles. El diseño de este lenguaje fue dispuesto así con la intención de reutilizarlo cada vez que se trate de generar cualquier código.

Finalmente, tenemos que tomar en cuenta, que cada línea de código genera una nueva línea en el archivo resultante, por lo cual se ha creado a manera de excepción una instrucción especial que nos permite cambiar este comportamiento y no realizar ningún salto de línea. Esta instrucción se escribe como #_#. Otra de las definiciones importantes, es que todos los datos son de tipo cadena.

5.1. Contexto de valores

LENCOS se encarga de manipular propiedades y la manera en que son inyectadas al intérprete es a partir de un contexto de valores. El contexto de valores es un *stream* con formato *XML* y la primer versión que llega al intérprete proviene del proceso que lo ha llamado.

Las operaciones de manipulación de las propiedades son la consulta y modificación de un valor, creación de una propiedad y recuperación de un sub-contexto de propiedades. La operación más usual es la consulta de un valor y se llama cada vez que el *script* requiera imprimir el valor de una propiedad. Otra es la recuperación de un sub-contexto, la cual se llama cuando tratamos con colecciones.

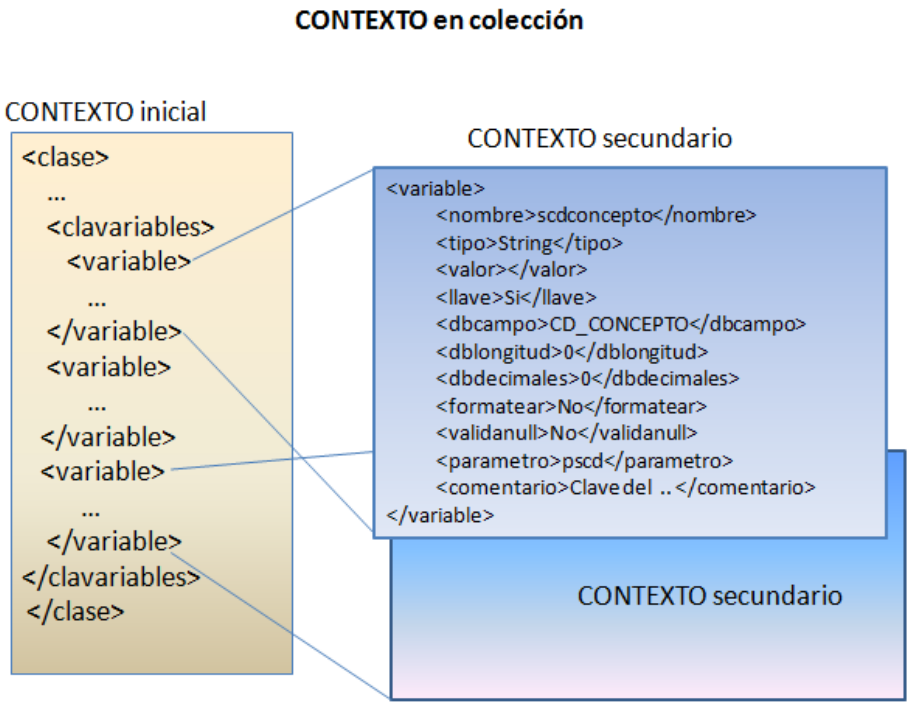

Figura 33. Arreglo de contexto de valores.

El diseño para manejar un sub-contexto de valores tiene los objetivos de simplificar la operación de consulta y permitir la creación de propiedades de ámbito local. En la Figura 33 se puede apreciar la creación de contextos en una instrucción que recorre una colección de propiedades llamadas *variable*. En las instrucciones de recorrido de colecciones se crea un nivel adicional cada vez que inicia una interacción, pero al final se destruye.

Para lograr la posibilidad de tener más de un contexto, el intérprete maneja una pila de contextos, de este modo, podemos tener tantos como sean requeridos. En el ejemplo de la Figura 33, se almacenarían en la pila sólo dos contextos. Sin embargo, es posible agregar a la pila tantos como se requieran.

El proceso de interpretación interactúa estrechamente con la pila, y no es raro encontrar más de un elemento cuando le ha tocado recorrer colecciones. Un ejemplo práctico sería cuando encontramos una instrucción que recorre una colección dentro de un recorrido previo y así sucesivamente.

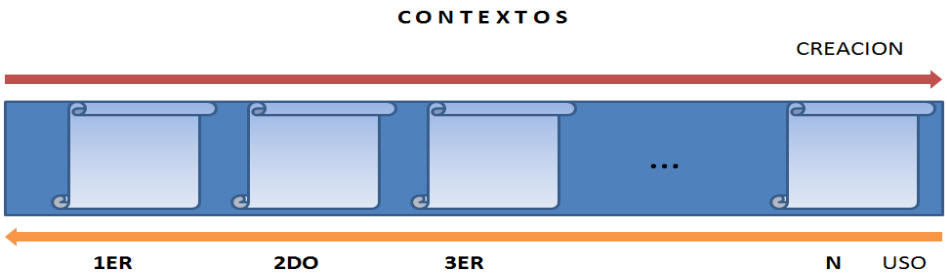

Figura 34. Creación y uso del arreglo de contextos.

Los contextos de valores no sólo sirven para delimitar el ámbito de valores con los cuales trabaja una iteración, también un contexto nos permite crear nuevas variables que funcionan sólo en el ámbito en que fueron creadas, y una vez que la iteración termina, en automático se elimina el contexto y por ende la variable.

Si la variable fue creada en un ámbito anterior y se le busca en el ámbito actual, el intérprete de *LENCOS* buscará primero si existe esta variable, iniciando con el contexto actual y si no se encuentra, se continúa en orden con los contextos anteriores, ver Figura 34.

5.2. Palabras claves del lenguaje

En el *LENCOS* tenemos las palabras clave de la Figura 35.

No.	Descripción	Instrucción
1.	Impresión de valores de las propiedades	#= ... #
2.	Ciclo de colección	# PARA CADA ... EN ... #
3.	Condición	# SI ... ESIGUALA ... #
4.	Validar colección	# SIHAY ... #
5.	Asignación de valores	# ASIGNA ... AVARIABLE ... #
6.	Contextualizando	# USANDO ... #
7.	Incremento	# INCREMENTA ... #

Figura 35. Instrucciones del lenguaje.

Además también tenemos las funciones para la manipulación de valores de la Figura 36.

Descripción	función
Conversión a minúsculas	aminuscula()
Conversión a mayúsculas	amayuscula()
Quitar espacios	trim()
Longitud	length()
Sub cadena	substring()

Figura 36. Funciones del lenguaje.

La sintaxis de *LENCOS* se definió utilizando la notación de Backus-Naur (*BNF*) y su definición completa se encuentra en el Apéndice A. A continuación se describen las instrucciones de la Figura 35 con más detalle.

5.2.1. Impresión de valores de las propiedades #= ...

Para lograr entender la impresión de valores de las propiedades, primero debemos entender el contexto de valores sobre el cual trabaja el intérprete de generación de código. El intérprete de generación de código tiene dos entradas, por un lado tenemos un contexto de valores dispuestos en formato *XML* y por el otro lado tenemos nuestro archivo de *LENCOS*. El resultado será el código generado por el intérprete. Un ejemplo del contexto de valores es el que se muestra en Figura 37.

```
<clase>
  <nbclase>Concepto</nbclase>
  <archclase></archclase>
  <commclase>Clase que tiene la información relativa a los conceptos</commclase>
  <nbentidadato>COB001_CONCEPTO</nbentidadato>
  <llavecompuesta>No</llavecompuesta>
  <clavariables>
    <variable>
      <nombre>scdinstitucion</nombre>
      <tipo>String</tipo>
      <valor></valor>
      <llave>Si</llave>
      <dbcampo>CD_INSTITUCION</dbcampo>
      <dblongitud>0</dblongitud>
      <dbdecimales>0</dbdecimales>
      <formatear>No</formatear>
      <validanull>No</validanull>
      <parametro>pscd</parametro>
      <comentario>Clave de la institución</comentario>
    </variable>
    <variable>... </variable>
</clavariables>
</clase>
```

Figura 37. Un ejemplo de un contexto de valores.

Y un ejemplo de un *script LENCOS* que respondería a este contexto de valores es el mostrado en la siguiente Figura.

```
1.  package #=clase.paquete.beans#;
2.
3.  /**
4.   * Descripción:
5.   * #=clase.commclase#
6.   * @creation date: (#=clase.general.fhcreacion#)
7.   * @author: #=clase.general.nbprogramador#
8.   * @conpany: #=clase.general.nbempresa#
9.   **/
10.
11. public class #=clase.nbclase# {
12.  #PARA CADA variable EN clase.clavariables#
13.    #SI variable.tipo ESIGUALA String#
14.        private #=variable.tipo# #=variable.nombre# = "";          // #=variable.comentario#
15.    #FIN SI#
16.    #SI variable.tipo ESIGUALA boolean#
17.        private #=variable.tipo# #=variable.nombre# = false;      // #=variable.comentario#
18.    #FIN SI#
19.    #SI variable.tipo ESIGUALA int#
20.        private #=variable.tipo# #=variable.nombre# = 0;           // #=variable.comentario#
21.    #FIN SI#
22.    #SI variable.tipo ESIGUALA double#
23.       private #=variable.tipo# #=variable.nombre# = 0;           // #=variable.comentario#
24.    #FIN SI#
25.    #SI variable.tipo ESIGUALA long#
26.      private #=variable.tipo# #=variable.nombre# = 0;           // #=variable.comentario#
27.    #FIN SI#
28. #FIN PARA CADA#
29.
30. /**
31.   * CONSTRUCTOR
32.   */
33. public #=clase.nbclase#() {
34.     super();
35. }
36.
37. }/* fin clase [#=clase.nbclase#] */
```

Figura 38. Un ejemplo de un script en LENCOS.

En la Figura 37 y 38, se pueden visualizar las dos entradas que recibe el intérprete de *LENCOS*. Por un lado tenemos el contexto de valores, y por el otro un *script* en *LENCOS* que será el encargado de realizar operaciones con el contexto de valores para lograr producir el código en el lenguaje seleccionado.

Es importante aclarar que la generación de código tendrá prácticamente todas las características del *script* en *LENCOS*, es decir, la salida respetará los textos y espacios que se tienen antes y después de los delimitadores # *instrucción* #. El intérprete sólo va a procesar las instrucciones de su interés y el resto se mantendrá intacto.

La ejecución del *LENCOS* lleva un control estricto del segmento de código que se está ejecutando, un segmento se determina por un número de línea inicial y un número de línea final. En el caso inicial de la ejecución de un *script*, el segmento inicia en la línea

1, y termina en la última línea del *script*. Pero para la ejecución de un grupo de instrucciones que pertenecen a un ciclo (por ejemplo), el segmento estará determinado por la primera línea después de la declaración del ciclo, y hasta la línea anterior al final del ciclo.

LENCOS interpreta grupos de instrucciones determinados con un apuntador de inicio, y un apuntador de fin. En el caso de requerir la ejecución de un sub-grupo de instrucciones, se llama recursivamente el proceso principal del intérprete de *LENCOS* para resolver el sub-grupo de instrucciones.

La impresión de propiedades puede ejemplificarse en la línea 11 del *script* en la Figura 38. La línea dice lo siguiente:

public class **#=clase.nbclase#** {

El intérprete procesa esta línea, considerando el contexto de valores definido en la Figura 37, por lo cual, una vez identificando que se trata de la impresión del valor de una propiedad, primero buscará y resolverá todo el contexto de *clase* y posteriormente en ese contexto buscará la propiedad *nbclase*, de la cual sabemos que es la última propiedad a buscar y de ahí resolverá su valor para imprimirlo.

La definición en *BNF* se encuentra en Figura 39.

<impresión-valor> ::= "=" [<expresión>]
<expresión> ::= [<contexto-id> | <función> "(" <contexto-id> { "," < número> } { "," < número> } ")"]
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>
<contexto> ::= {<identificador>}
<identificador> ::= [<letra>]{ <letra> | <digito> | "_" | "-" }

Figura 39. BNF de asignación.

La expresión regular asociada a las asignaciones es la siguiente:

#=(?<**name**>\w+|\w+\.\w+|\w+\.\w+\.\w+|\w+\.\w+\.\w+.\w+)#

Con esta expresión regular, el intérprete de *LENCOS* realiza una asociación de la expresión regular contra la línea del *LENCOS*, y recupera la propiedad ***<name>*** para resolverla en el contexto de valores. Como se podrá ver, el máximo de niveles de propiedades de búsqueda es de cuatro.

Además de esta expresión regular, también se usa otra para lograr identificar las funciones del *LENCOS*.

#=(?**<funcion>**\w+)\((?**<name>**\w+|\w+\.\w+|\w+\.\w+\.\w+|\w+\.\w+\.\w+.\w+)(?**<par1>**,\w+)?(?**<par2>**,\w+)?\)#

Y en esta ocasión del mismo modo se identifica la propiedad **<name>**, pero también las propiedades **<funcion>**, **<par1>** y **<par2>**, estos dos últimos son opcionales.

Una vez que se han recuperado los valores, el intérprete de *LENCOS* procede en primer lugar a resolver lo relacionado con el contexto de valores, y posteriormente se procesa ese valor con la función que fue identificada.

5.2.2. Ciclo # PARA CADA … EN …

Un ciclo de esta naturaleza recorre cada una de las propiedades pertenecientes a una colección de instrucciones. Por cada propiedad de una colección se procesa el conjunto de instrucciones que se encuentra dentro del ciclo. El grupo de instrucciones dentro del ciclo, puede ser cualquiera que pertenezca al *LENCOS*.

En cuanto al ciclo, tenemos dos tipos de ciclo:

1. El ciclo sencillo, # **PARA CADA** … **EN** … #
2. Y el ciclo condicionado # **PARA CADA** … **EN** … **DONDE** … **CONDICION** … #

En donde lo que identificamos como CONDICION puede tener dos valores fijos "ESIGUALA" o "ESDIFERENTEA".

La definición en *BNF* se encuentra en Figura 40.

```
<instrucción-ciclo> ::= "PARA CADA "  <identificador> " EN "  <contexto-id> {" DONDE "
<identificador> ["ESIGUALA" | "ESDIFERENTEA"] <constante>} [<instrucciones>] "FIN PARA CADA"
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>
<identificador> ::= [<letra>] { <letra> | <digito> | "_" | "-" }
<constante> ::= [<letra> | <digito>] {<letra> | <digito>}
<instrucciones> ::= [<instrucción> | <línea-impresión>] {<instrucción> | <línea-impresión>}
```

Figura 40. BNF del ciclo.

Un ejemplo práctico de los ciclos lo podemos encontrar en la línea 12 de la Figura 38.

12. #**PARA CADA** variable **EN** clase.clavariables#

Una vez que el intérprete de *LENCOS* llegue a esta posición, el contexto que cobrará importancia será como el de la Figura 41, donde el proceso de generación segmentará en cada pasada del ciclo un sub-contexto recursivo de la propiedad variable de esa posición. El llamado sub-contexto recursivo se ha diagramado antes en la Figura 34.

```
<clase>
   ...
   <clavariables>
      <variable>
        ...
      </variable>
      <variable>
        ...
      </variable>
      <variable>
        ...
      </variable>
   </clavariables>
</clase>
```

Figura 41. Segmentación interna en un ciclo.

El contexto de valores es un flujo de caracteres que contiene todas las propiedades que son consideradas por el intérprete de *LENCOS*. En el caso particular de una instrucción de ciclo, se crea un arreglo de contextos, dependiendo del número de ciclos anidados que existan. Estos arreglos de contextos serán considerados por el intérprete de *LENCOS* para recuperar los valores de las propiedades solicitadas. En el caso de que tengamos *n* contextos en el arreglo, el primer contexto a ser considerado será el contexto *n*, y en caso de no encontrar el valor de la propiedad requerido, el intérprete seguirá buscando en el contexto *n-1*.

Las dos expresiones regulares asociadas a los ciclos son las siguientes:

```
#PARA CADA\s+(?<name>\w+)\s+EN\s+(?<coleccion>\w+|\w+\.\w+|\w+\.\w+\.\w+|\w+\.\w+\.\w+.\w+)\s*#
```

```
#PARA CADA\s+(?<name>\w+)\s+EN\s+(?<coleccion>\S+)\s+DONDE\s+(?<variable>\S+)\s+(?<condicion>ESIGUALA|ESDIFERENTEA)\s+(?<constante>\S+)\s*#
```

5.2.3. # SI … ES IGUALA …

El condicionamiento en un *script* podría ser simple o compuesto, el condicionamiento simple tiene la siguiente forma:

SI … **ESIGUALA**|**ESDIFERENTEA**|**ESTAEN** …
…*grupo de instrucciones*…
#**FIN SI**#

Y el compuesto:

SI … **ESIGUALA**|**ESDIFERENTEA**|**ESTAEN** …
…*grupo de instrucciones*…
#**Y SI NO**#
…*grupo de instrucciones*…
#**FIN SI**#

En el condicionamiento, el intérprete de *LENCOS* primero resolverá las solicitudes de valores de las propiedades en el contexto y posterior a ellos segmentará el código dependiendo de si la condición se cumplió o no. Cabe destacar que esta instrucción de condicionamiento no crea un nuevo contexto de valores.

La definición en *BNF* se encuentra en Figura 42.

```
< instrucción-si-condición>  ::= "SI "<contexto-id>  ["ESIGUALA"|"ESDIFERENTEA"|"ESTAEN"]
[<constante>| <identificador>] [<instrucciones>] "FIN SI"
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>
<identificador> ::= [<letra>]{ <letra> | <digito> | "_" | "-" }
<constante> ::= [<letra> | <digito>]{<letra> | <digito>}
<instrucciones> ::= [<instrucción> | <línea-impresión>]{<instrucción> | <línea-impresión>}
```

Figura 42. BNF de condición.

La expresión regular de la condición es:

#**SI**\s+(?<name>\S+)\s+(?<condicion>**ESIGUALA**|**ESDIFERENTEA**|**ESTAEN**)\s+(?<constante>=?\S+)\s*#

5.2.4. # SIHAY ...

La instrucción es una condición que determina si un conjunto de elementos está formado por al menos un elemento. La instrucción tiene dos variantes, una que valida la existencia de elementos dentro de un conjunto SIN CONDICIONES, y la segunda variante que valida la existencia de elementos dentro de un conjunto, pero después de haber aplicado un condicionamiento a cada elemento. La instrucción simple es como sigue.

#**SIHAY** … **EN** … #
    …*grupo de instrucciones*…
#**FIN SIHAY**#

Esta instrucción se usa, cuando se quiere determinar si una propiedad que tiene un arreglo de propiedades en contexto de valores, tiene al menos una propiedad de este arreglo. En la siguiente Figura se muestra un ejemplo.

CONTEXTO	LENCOS
<clase> <clavariables> </clavariables> </clase>	#SIHAY variable EN clase.clavariables# *…grupo de instrucciones…* #FIN SIHAY#

Figura 43. Ejemplo de instrucción SIHAY.

Ingresando el ejemplo descrito en la Figura 43 al intérprete de *LENCOS*, tendremos que el resultado es falso, por lo cual no se procederá a ejecutar el conjunto de instrucciones que se encuentran en el cuerpo de la instrucción SIHAY. La instrucción con condición tiene la siguiente forma.

#**SIHAY** … **EN** … **DONDE** … **ESIGUALA** …#
…
#**FIN SIHAY**#

La instrucción "**SIHAY**" complementada con una condición, permitirá determinar si al menos se encuentra un elemento de una colección de elementos pero inmediatamente después de considerar una condición que es aplicada directamente a las propiedades de cada elemento del arreglo. En la siguiente Figura se muestra un ejemplo.

CONTEXTO	LENCOS
<clase> <clavariables> <variable> <nombre>scdinstitucion</nombre> <tipo>String</tipo> <llave>Si</llave> </variable> <variable> <nombre>scdconcepto</nombre> <tipo>String</tipo> <llave>Si</llave> </variable> <variable> <nombre>snbconcepto</nombre> <tipo>String</tipo> <llave>No</llave> </variable> </clavariables> </clase>	#SIHAY variable EN clase.clavariables DONDE variable.llave ESIGUALA Si# *…grupo de instrucciones…* #FIN SIHAY#

Figura 44. Ejemplo de instrucción SIHAY condicional.

Ingresando el ejemplo descrito en la Figura 44 al intérprete *LENCOS*, tendremos que el resultado es verdadero, por lo cual se procederá a ejecutar el conjunto de instrucciones que se encuentran en el cuerpo de la instrucción SIHAY.

La definición en *BNF* se muestra en Figura 45.

```
< instrucción-si-hay> ::= "SI HAY "<identificador> " EN "  <contexto-id> {" DONDE " <identificador> ["ESIGUALA"
| "ESDIFERENTEA"] <constante>} [<instrucciones>] "FIN SI HAY"
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>
<identificador> ::= [<letra>]{ <letra> | <digito> | "_" | "-" }
<constante> ::= [<letra> | <digito>]{<letra> | <digito>}
<instrucciones> ::= [<instrucción> | <línea-impresión>]{<instrucción> | <línea-impresión>}
```

Figura 45. BNF de condición SIHAY.

La expresión regular de la instrucción es:

```
#SIHAY\s+(?<name>\w+)\s+EN\s+(?<coleccion>\w+|\w+\.\w+|\w+\.\w+\.\w+)\s*#
```

Y

```
#SIHAY\s+(?<name>\w+)\s+EN\s+(?<coleccion>\S+)\s+DONDE\s+(?<variable>\S+)\s+
ESIGUALA\s+(?<constante>=?\w+|=?\w+\.\w+|=?\w+\.\w+\.\w+)\s*#
```

5.2.5. # ASIGNA … AVARIABLE …

La asignación de valores es una de las dos instrucciones que afectan el contexto de valores, esto es, integra una propiedad y su valor respectivo en un grupo de propiedades.

La asignación solamente va a integrar la pareja de (propiedad, valor), en el último contexto creado por el intérprete *LENCOS*. Con esto aseguramos que el ámbito de una propiedad pertenezca al contexto donde fue creada. Si creamos una propiedad en un ámbito exterior, su valor estará disponible en cualquier ámbito interior. Lo anterior se resuelve de forma automática por la forma en cómo es ejecutada la impresión de un valor por la instrucción #=…#.

Una de las ventajas de crear/asignar una pareja (propiedad, valor) en el contexto de valores actual, es que una vez que dejamos un subconjunto de instrucciones, la característica retráctil del contexto de valores automáticamente desaparece el último contexto y con ello la pareja (propiedad, valor) que previamente creamos. La definición en *BNF* se encuentra en Figura 46.

<instrucción-asignación> ::= "ASIGNA" <constante> "AVARIABLE" <contexto-id>
<constante> ::= [<letra> | <digito>]{<letra> | <digito>}
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>

Figura 46. BNF de ASIGNA.

La expresión regular de la instrucción es:

#ASIGNA\s+(?<constante>=?\w+|=?\w+\.\w+|=?\w+\.\w+\.\w+)\s+AVARIABLE\s+(?<variable>\w +)\s*#

5.2.6. # USANDO ...

La instrucción de contextualización, crea un nuevo contexto de valores considerando la propiedad de un arreglo de propiedades y en una posición en particular. Una vez determinado este nuevo contexto, todas las instrucciones que se encuentren en el cuerpo de la instrucción van a considerar este contexto de valores como el último contexto desde donde extraerán sus valores.

La instrucción con condición tiene la siguiente forma:

#**USANDO** … **EN** … **POSICION** … #
…
#**FIN USANDO**#

La siguiente Figura muestra un ejemplo.

CONTEXTO	**LENCOS**
<clase> <clavariables> <variable> <nombre>scdinstitucion</nombre> <tipo>String</tipo> <llave>Si</llave> </variable> **<variable>** **<nombre>scdconcepto</nombre>** **<tipo>String</tipo>** **<llave>Si</llave>** **</variable>** <variable>	#USANDO variable EN clase.clavariables POSICION **2** # *…grupo de instrucciones…* #FIN USANDO#

```
      <nombre>snbconcepto</nombre>
      <tipo>String</tipo>
      <llave>No</llave>
   </variable>
</clavariables>
 </clase>
```

Figura 47. Ejemplo de USANDO.

El ejemplo de la Figura 47 crea un nuevo contexto de valores tomando en consideración sólo lo que está marcado en **negritas**.

La definición en *BNF* se encuentra en Figura 48.

```
< instrucción-usando> ::= "USANDO" <identificador> "EN"  < contexto-id> "POSICION"
<número> [<instrucciones>] "FIN USANDO"
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>
<instrucciones> ::= [<instrucción> | <línea-impresión>] {<instrucción> | <línea-impresión>}
<identificador> ::= [<letra>] { <letra> | <digito> | "_" | "-" }
<número> ::= <digito> | <número> <digito>
```

Figura 48. BNF de USANDO.

La expresión regular de la instrucción es:

```
#USANDO\s+(?<name>\w+)\s+EN\s+(?<coleccion>\w+|\w+\.\w+|\w+\.\w+\.\w+)\s+POSICION\s+(?<numero>\d+)\s*#
```

5.2.7. # INCREMENTA ...

La instrucción de incremento realiza una suma a un número entero guardado en una propiedad del arreglo de contextos de valores.

La definición en *BNF* se encuentra en Figura 49.

```
<instrucción-incremento> ::= "INCREMENTA" <constante> "AVARIABLE" <contexto-id>
<constante> ::= [<letra> | <digito>] {<letra> | <digito>}
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>
```

Figura 49. BNF de INCREMENTA.

La expresión regular de la instrucción es:

```
#INCREMENTA\s+(?<constante>=?\w+)\s+AVARIABLE\s+(?<variable>\w+)\s*#
```

5.3. Interpretación de script LENCOS

El intérprete de *LENCOS* es un proceso que se dedica a generar el código correspondiente en alguna sintaxis destino. Estas sintaxis deben ser previamente especificadas en el archivo de configuración que incluye la meta-arquitectura y el meta-lenguaje. La configuración incluye los apartados o directorios que agrupan todos los componentes que pertenecen a una meta-arquitectura. Los componentes que se procesan, son *scripts* codificados en *LENCOS*.

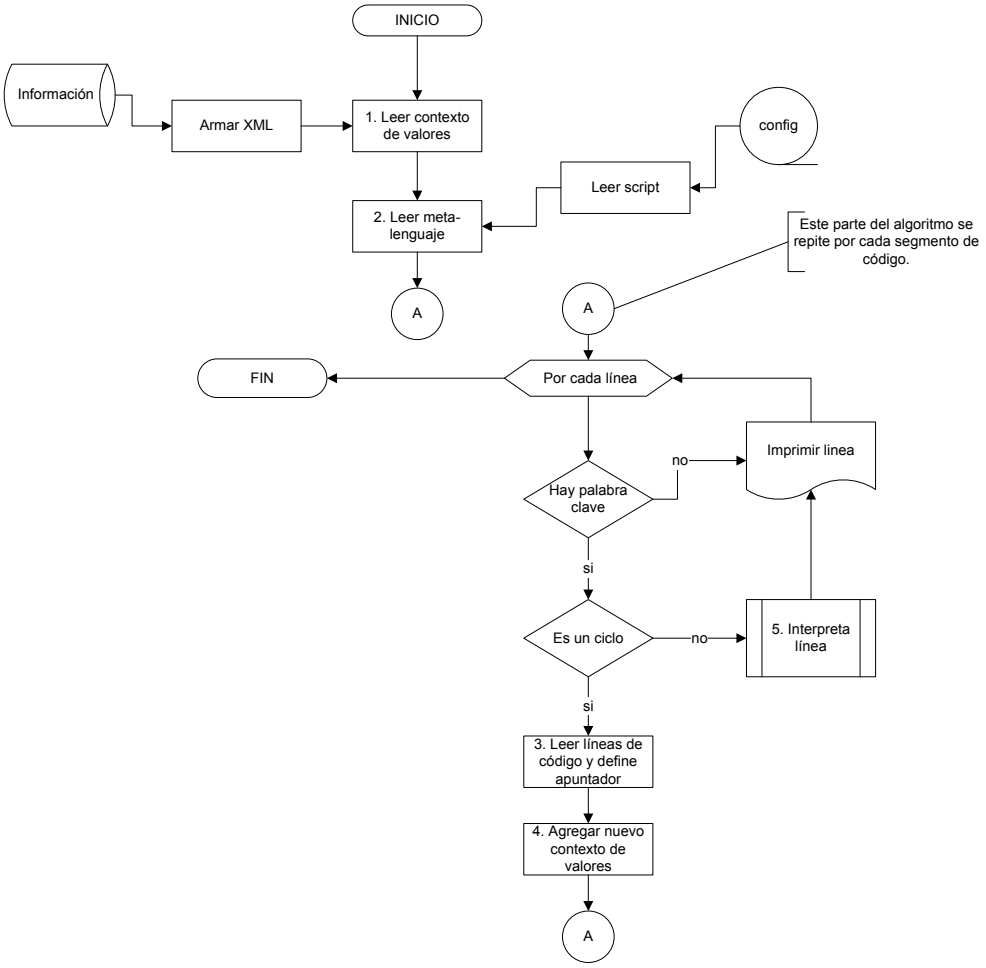

Figura 50. Algoritmo de interpretación de un script.

El algoritmo de interpretación de un *script* en Figura 50, inicia con el proceso "*1. leer el contexto de valores"*, que se encarga de consultar la base de datos de la herramienta donde se tiene la información de los proyectos. Esta información es manipulada para armar un flujo (*stream*) en *XML* con una agrupación jerárquica, cada agrupación tendrá las propiedades y sus valores correspondientes (*ver Apéndice E – Sección 1*).

El intérprete continúa con la operación "*2. Leer meta-lenguaje"*, que consiste en leer el *script* del componente que se desea generar. Este componente debe estar codificado en *LENCOS*. Cada línea del *script* es cargada secuencialmente en un arreglo de memoria, el cuál nos permite acceder a cada línea por su posición. La posición de la línea es relevante, ya que internamente se usan apuntadores de inicio, fin y línea actual para controlar la ejecución del *script*.

Hay un sub-proceso de interpretación de un segmento de código. El segmento inicial siempre es el *script* completo con un apuntador de inicio igual a la línea uno, y un apuntador final a la última línea. Cada vez que se encuentra un segmento de código (agrupado por una instrucción), el sub-proceso se llama recursivamente para procesar el segmento y escribir el resultado en el archivo final.

La lectura de un segmento de código se ilustra en la *operación 3*, de la Figura 50. Esta operación busca la línea inicial, final y define sus apuntadores. En la *operación 4*, se crea un nuevo contexto de valores para poder crear variables y asignarles sus valores. El contexto de valores es el ámbito en el que están disponibles las variables y sus valores creados en ese segmento de código.

Hay variables que son creadas *al vuelo* durante el recorrido de un arreglo, estas variables son: "*esprimero*" y "*esultimo*". Estas dos variables nos permiten identificar cuando un elemento se encuentra en la primer o última posición del arreglo. Esto facilita y deja clara la codificación, además que es muy útil durante la interpretación del *script LENCOS*.

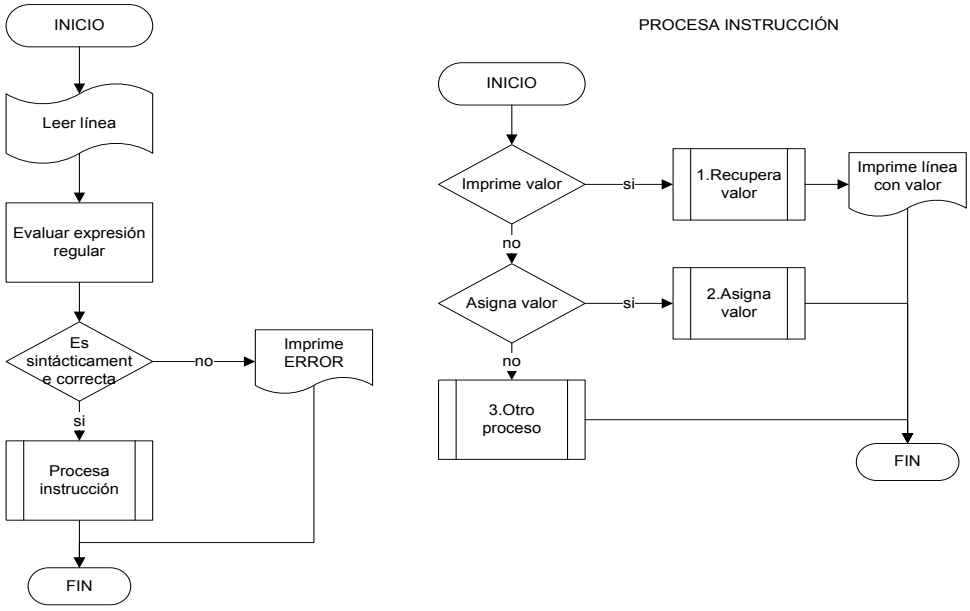

Figura 51. Algoritmo de interpretación de una instrucción.

El algoritmo de interpretación de una línea recibe un arreglo de contexto de valores, el tamaño del arreglo depende del número de segmentos de código que se estén ejecutando. El arreglo de contextos participa cuando se pretende recuperar el valor de una propiedad (Figura 51*, operación 1*), y el procedimiento siempre inicia buscando la propiedad en cada contexto en el sentido inverso en que fueron creados, una vez que la propiedad ha sido encontrada, se recupera el valor y se termina la búsqueda.

En el lenguaje, también se permite crear variables para conservar algunos valores dentro del ámbito actual. Estas nuevas variables y sus respectivos valores son creados en el último esquema de valores (Figura 51*, operación 2*).

Se ha previsto que el lenguaje también pueda realizar algunas otras operaciones y para ello se ha dejado el *proceso 3* de la Figura 51.

EJEMPLO

La ejecución de un *script LENCOS* se logra usando dos apuntadores, un apuntador de inicio de instrucción y un apuntador final. La primera vez que se ejecuta un *script*, el apuntador de inicio apunta a la primera línea y el apuntador final a la última línea de código. Las llamadas recursivas sólo se hacen en el momento en que se encuentra una instrucción que implique la ejecución de un grupo de instrucciones.

Figura 52. Llamadas recursivas del proceso de interpretación.

Las instrucciones que generan llamadas recursivas del proceso de interpretación son

PARA CADA … EN … #, # SI … ESIGUALA … #, # SIHAY … #, # USANDO … #.

En el ejemplo de la Figura 52, tenemos que los primeros apuntadores son aquéllos de color azul, es entonces cuando el proceso de interpretación se hace responsable de ejecutar cada instrucción que se encuentra entre esta pareja de apuntadores. En el momento en que identifica una instrucción que amerita la llamada recursiva del proceso se marca como un apuntador inicial y se busca su respectivo apuntador final. Ya que se tiene la pareja de apuntadores, entonces se procede a llamar el proceso de interpretación otra vez.

CAPÍTULO IV

CASO DE ESTUDIO

1. INTRODUCCIÓN

En este capítulo vamos a demostrar el uso de la herramienta en un sistema de aula virtual. La arquitectura de implementación es *CGI (Common Gateway Interface) y* la generación de código es en *PHP*. En este capítulo vamos a considerar las entidades de datos y la interfaz del usuario más representativa del sistema ya que las demás funciones tienen un tratamiento muy similar.

En la introducción del caso de estudio se explica el ambiente general en el que se desarrolla el sistema y los módulos que lo integran, posteriormente se explican los insumos de la herramienta, es decir, el modelo *entidad-relación* y el prototipo.

De acuerdo a la propuesta del método de generación, las entidades de datos se integran en un *script* con una sintaxis *DDL* (*Data Definition Language*). Sin embargo, en el ejemplo, sólo se va a considerar el *script* de una entidad para demostrar el proceso de integración y generación del código.

De acuerdo al método, después que se genera el código *Back-End*, se inicia una etapa de adaptación del código. La etapa de adaptación, permite implementar la lógica de negocio de forma artesanal apoyándose en el código que se generó automáticamente. Esta etapa es independiente de la herramienta y no forma parte del alcance de la explicación.

Una vez que se logró tener una lógica de negocio estable, entonces se procede a la integración con el prototipo y la generación de código del *Front-End*. El proceso de integración, es un proceso operado desde la interfaz de la herramienta, donde se relacionan las pantallas con los componentes de software. Finalizando la integración se procede a generar el código.

Es importante destacar, que en este capítulo se demuestran los procedimientos de integración y generación de código *Front-End*, considerando solamente una página representativa de la nueva aplicación. Incluso, el proceso de integración solo tomará en cuenta un método, un campo y un evento.

Desde el primer momento que se tiene capturada la información del sistema, es posible automatizar la documentación, y a partir de entonces, se podrá generar cada documento en una versión actualizada y definitiva.

2. INTRODUCCIÓN AL CASO DE ESTUDIO

El producto que vamos a utilizar para demostrar la tesis, tiene que ver con un sistema de aprendizaje vía Internet. Este sistema nos permitirá tener una o más aulas virtuales, además de varios eventos relacionados. Los eventos podrán ser tipo 1. Examen, 2. Cuestionario, 3. Recursos institucionales y 4. Recursos del profesor, ver Figura 53.

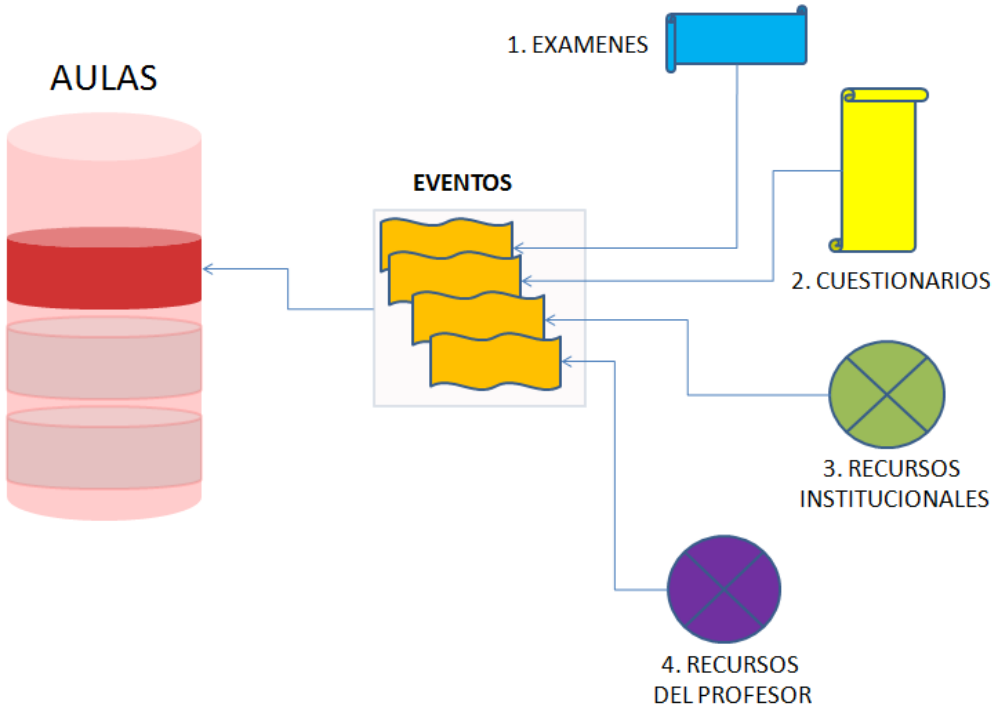

Figura 53. Las entidades generales.

De acuerdo con la metodología de la Figura 55 y explicada en el capítulo III, lo primero que se hace es concluir la etapa de análisis y diseño del sistema. Algunos de los resultados de la etapa de diseño son los insumos principales, es decir, el diagrama *entidad-relación* y el prototipo de la nueva aplicación.

Las funciones y pantallas de la Figura 54, que se consideran para el caso de estudio son:

1. La creación del aula virtual permitirá crear nuestra entidad principal para contener ahí los temas y recursos académicos que estarán disponibles para los alumnos.
2. Los temas y sus recursos son documentos, ligas, videos o audios integrados a los cursos.

3. La configuración del contenido para cada aula permitirá organizar los recursos y tareas de forma cronológica para los alumnos.
4. El proceso cognoscitivo es la pantalla desde donde el alumno interactuará con el aula virtual.

Figura 54. El Flujo de la Información.

En el sistema, los procesos 1, 2 y 3 son definidos por un profesor y en el proceso 4 interactúa el profesor y el alumno en una interfaz con funciones muy similares pero por separado.

2. EL MÉTODO DE GENERACIÓN

De acuerdo al método de construcción de la Figura 55, se tienen que diseñar por anticipado el modelo entidad-relación y el prototipo del caso de estudio. Los insumos deben ser integrados en dos archivos *scripts* por separado; el script del modelo entidad-relación, se integraría con formato de definición de datos (*DDL*), y el prototipo de la interfaz del usuario, debe incorporarse en un archivo HTML.

.

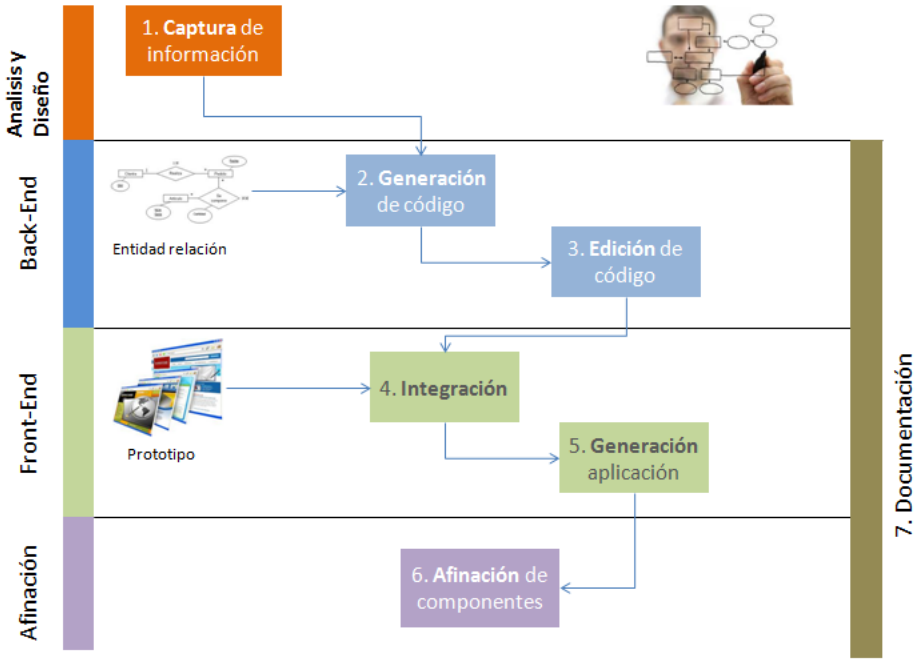

Figura 55. Método de generación de sistemas.

El diseño entidad-relación del caso de estudio, se puede ver en la Figura 56, y un ejemplo del *script* se puede ver en la Figura 63; el *script* es el resultado de una función de exportación de la herramienta de diseño de base de datos[1].

Un ejemplo del *script* con el HTML se puede ver en la Figura 57. El script es resultado de una etapa de diseño gráfico utilizando algún software especializado y modificado posteriormente para incluir unos identificadores relevantes para la herramienta[2].

[1] En este caso se utilizó Erwin 4.0 de la compañía Computer Associates International, Inc.
[2] En este caso se utilizó un Editor de texto comun.

3. MODELADO ENTIDAD-RELACIÓN A CONSIDERAR

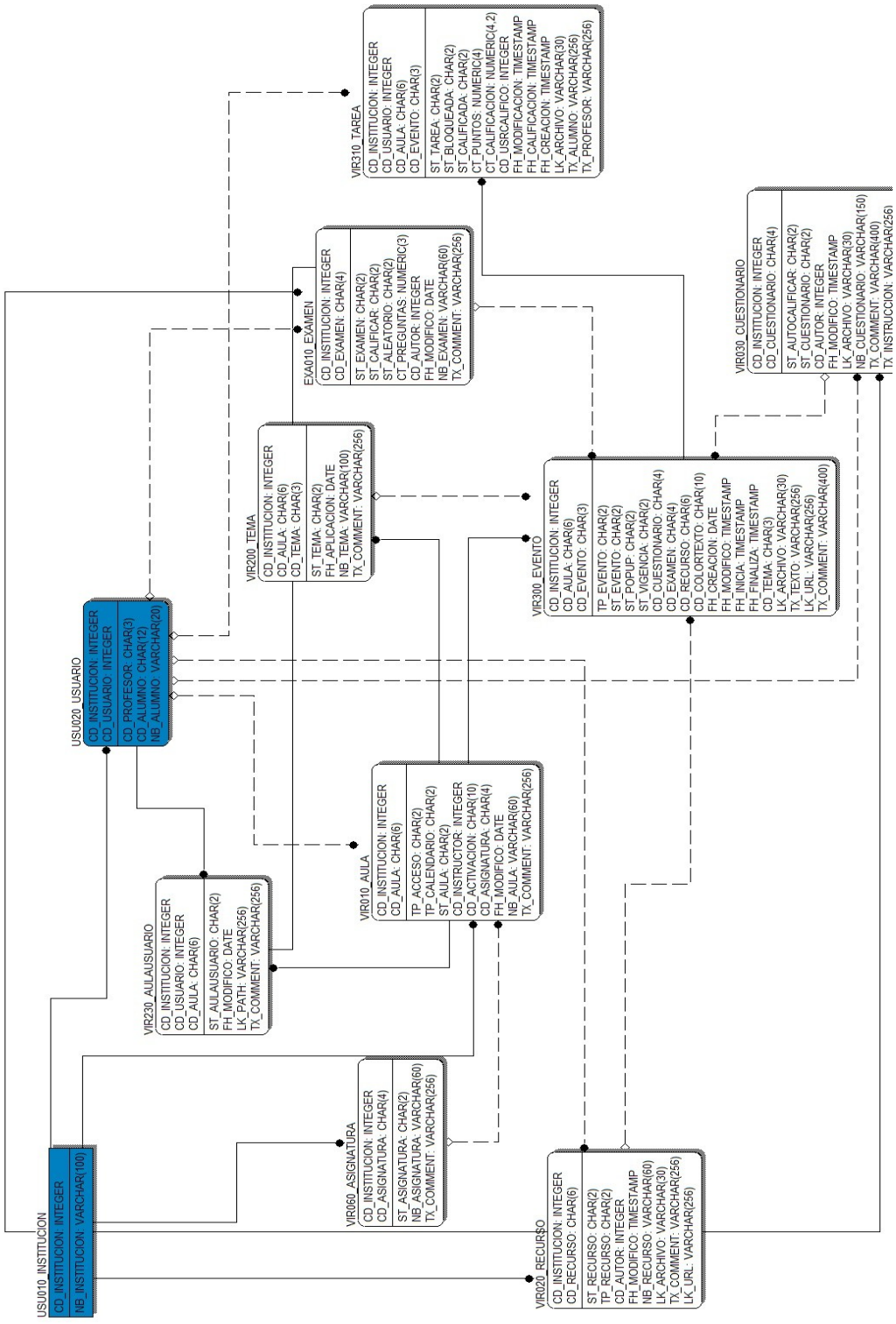

Figura 56. El Modelo entidad-relación del Caso de Estudio.

El diseño de la base de datos de la nueva aplicación considera el detalle de todas las entidades participantes. Estas entidades se van a crear en MySQL. La Figura 56, considera cada uno de los campos y sus respectivos tipos de datos.

Las aulas virtuales pertenecen a una institución y por ello la llave primaria *CD_INSTITUCION* se encuentra en cada una de las entidades del diagrama. Fuera de esa consideración, la Figura 56, es el diagrama *entidad-relación* de las entidades que se dibujaron en la Figura 53.

Los perfiles de profesores y alumnos son resumidos en claves del tipo entero, en llaves foráneas como *CD_INSTRUCTOR* y *CD_AUTOR*. La entidad con los nombres de profesores y alumnos no está explícitamente en el diagrama *entidad-relación*. Tampoco se encuentra diseñado el módulo de seguridad de la nueva aplicación.

La integración del *script* de la base de datos es el insumo requerido en la Figura 55, *paso 2 – Generación de código Back-End*.

4. PROTOTIPO A CONSIDERAR

Este prototipo es una propuesta para probar la herramienta en el proceso de integración y generación del código *Front-End*. Dispone de muchas pantallas para la operación del sistema, Pero para probar la herramienta, sólo vamos a detallar el proceso sobre una pantalla en particular.

Este prototipo es resultado de un análisis y diseño en las etapas previas a la construcción. El diseño comprende las funciones específicas que el usuario requiere para responder a sus necesidades. Es de esperarse, que cuando el prototipo llega a la herramienta, tendría que tener la aprobación del usuario.

La construcción del prototipo, suele ser responsabilidad de los expertos en diseño gráfico; aunque también puede ser producto de ingenieros de sistemas que implementan plantillas pre-diseñadas. En cualquiera de los casos, la herramienta espera un *script HTML* por cada interfaz o pantalla del usuario.

En la herramienta, cada página se maneja de forma independiente, y cada una se debe configurar en un proceso independiente en la etapa de integración. Al concluir la integración de cada pantalla, La herramienta valida que todos los controles o etiquetas se relacionen.

```
<form name="frmbuscar" method="post" enctype="multipart/form-data">
<input type="hidden" name="hidopcion" value = "">

<!--INCLUIR_JAVASCRIPT-->
<tr>
    <td>
        <table width="100%" border="0" cellspacing="0" cellpadding="0">
            <tr>
                <td width="5%"> </td>
                <td><p class="dato">INSTITUCION</p></td>
                <td colspan="4">
                   <select name="cboinstitucion" id="cboinstitucion" class="tx_gral1">
                      <option value="01">CIEX 1</option>
                   </select>
                </td>
            </tr>
            <tr>
                <td colspan="6"> </td>
            </tr>
            <tr>
                <td colspan="6" class="cabeceratitulo"><p class="titulotabla"><b>CRITERIOS DE
            </tr>
            <tr>
                <td width="5%"> </td>
                <td><p class="dato">Nombre</p></td>
                <td><input name="txtnombre" id="txtnombre" type="text" class="tx_gral1" size="
                <td width="5%"> </td>
                <td><p class="dato">No. Empleado</p></td>
                <td><input name="txtnoempleado" id="txtnoempleado" type="text" class="tx_gral1
            </tr>
        </table>
    </td>
</tr>
<tr>
   <td>
        <table border="0" cellspacing="0" cellpadding="0" width="100%">
            <tr>
                <td class="cabeceratitulo" colspan="11"><p class="titulotabla" align="center"><
            </tr>
            <tr>
                <td align="center" class="cabecera"><p class="textablacabecera">Nombre Completo
                <td align="center" class="cabecera"><p class="textablacabecera">Clave Profesor<
                <td align="center" class="cabecera"><p class="textablacabecera">Número Empleado
                <td align="center" class="cabecera"><p class="textablacabecera">RFC</p></td>
                <td align="center" class="cabecera"><p class="textablacabecera"></p></td>
            </tr>
            <tr id="trrenglones">
                <td align="center" class="fila" nowrap><p class="dato2" id="lblstnombre">Juan M
                <td align="center" class="fila" nowrap><p class="dato2" id="lblstclave">02Sep19
                <td align="center" class="fila" nowrap><p class="dato2" id="lblstempleado">23o4
                <td align="center" class="fila" nowrap><p class="dato2" id="lblstrfc">MANOM013
                <td align="center" class="fila" nowrap><a href="javascript:modificar('1','1');'
            </tr>
        </table>
   </td>
</tr>
```

Figura 57. Modelo de codificación para páginas HTML.

La identificación de controles y etiquetas se realiza por medio de una propiedad en cada *TAG* llamada *id*. Además, cada página *HTML* debe identificar la posición donde se insertan los segmentos de código. En la Figura 57 podemos ver un ejemplo de codificación.

La integración del *script HTML* del prototipo de la interfaz del usuario, es el insumo requerido en el la Figura 55, *paso 4 – Integración*.

4.1. Creación de una nueva aula

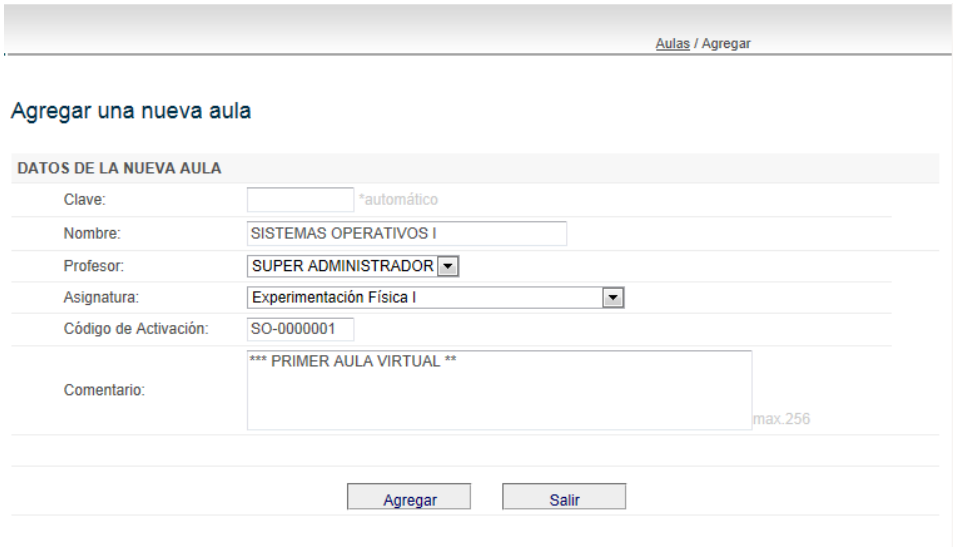

Figura 58. Nueva Aula Virtual.

Esta página nos permite agregar un aula virtual en el sistema, las aulas virtuales se asignan a un profesor y una asignatura; además, tienen un código de activación para que el alumno pueda integrarla como parte de su área de trabajo, ver Figura 58.

4.2. Consulta de aulas virtuales

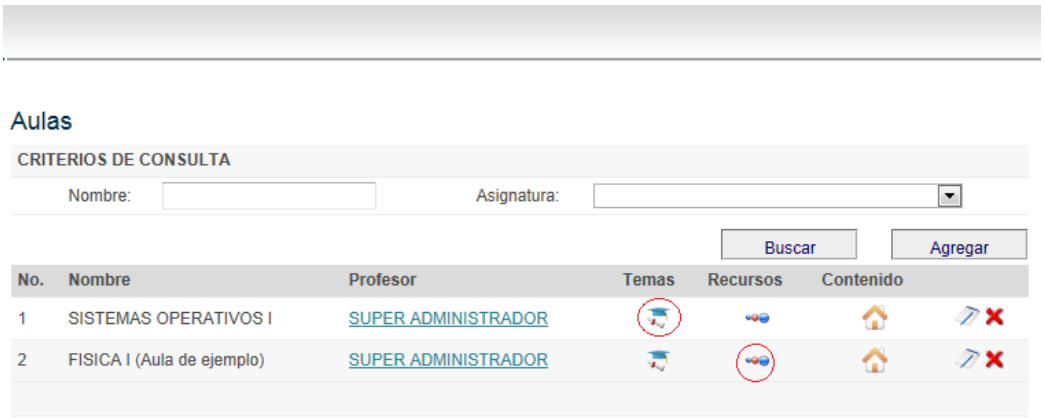

Figura 59. Consultando Aulas Virtuales.

Esta página nos permite realizar una consulta de las aulas virtuales registradas en el sistema, además nos permite integrar sus temas académicos y los recursos institucionales (documentación académica), ver Figura 59.

4.3. Temas de un aula

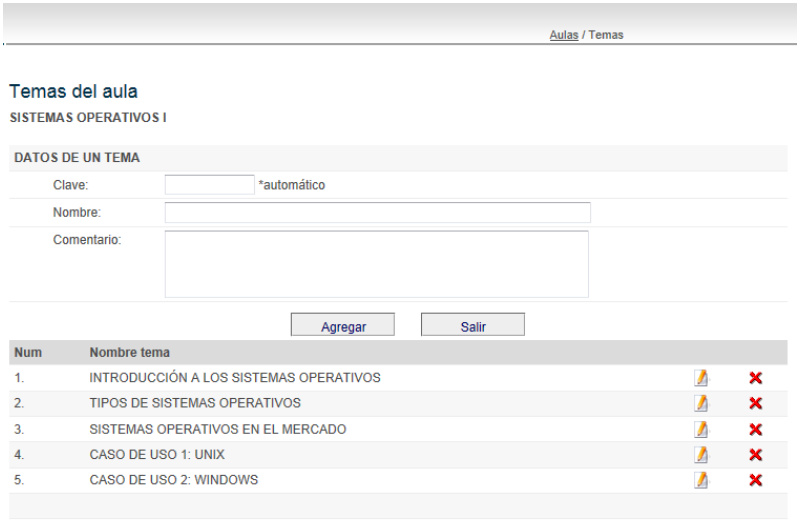

Figura 60. Temas del Aula Virtual.

Esta página nos permite agregar los temas académicos que son tratados en cada aula virtual. Los temas son los capítulos o secciones de la organización del contenido académico. Como ejemplo, estos temas pueden ser: meses, bimestres, módulos, capítulos, etc. Ver Figura 60.

4.4. Configuración de contenido

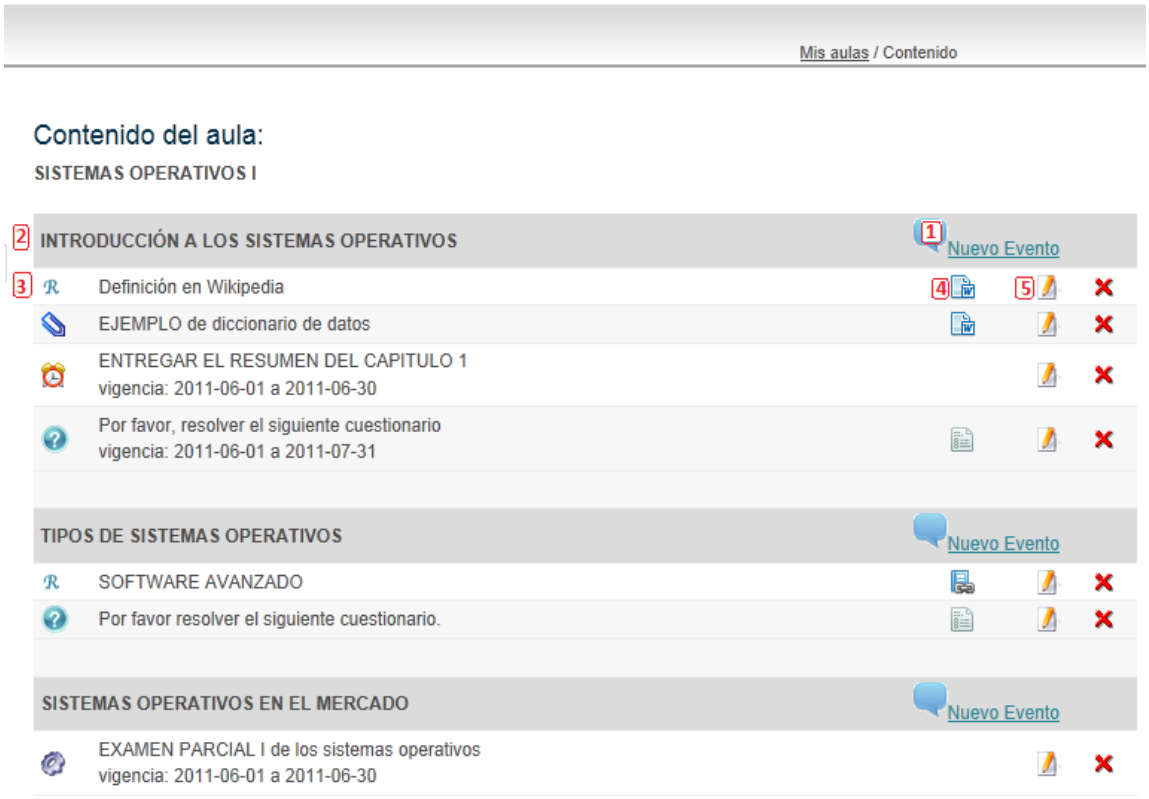

Figura 61. Configuración del Contenido.

Esta página de la Figura 61, tiene la función de administrar el contenido de estudio para un aula virtual en particular. Y está formada por lo siguiente:

1. Una liga para incorporar un nuevo evento a un tema.
2. La división de cada uno de los temas.
3. El tipo de evento que pertenece a un tema.
4. El tipo de documento (en caso de tener un archivo adjunto).
5. La liga para la modificación de un evento.

4.5. Interfaz entre Alumno-Profesor

Figura 62. Interfaz de Contenido.

Esta página de la Figura 62, tiene la misma funcionalidad que la página de la Figura 61, a excepción de que para los alumnos no es posible agregar o modificar un evento.

7. GENERACIÓN DEL BACK END

La generación de código es el producto de los pasos dos y cinco del método de generación de la Figura 55. Primero se genera el código *Back-End* y posteriormente el código *Front-End*. Sin embargo, es perfectamente valido terminar la línea de producción a partir del paso dos*;* esta excepción, permite utilizar el código *Back-End* en la construcción de aplicaciones que carecen de una interfaz de usuario. Aun así, la aproximación de la parte *Back-End* representa un gran porcentaje de aproximación de la aplicación final.

Este proceso *Back-End* considera como entrada la especificación en el lenguaje de definición de datos (*DDL, por sus siglas en inglés*) y una descripción de cada campo. Esta información se registra en la herramienta con el fin de utilizarlos en la documentación.

Cabe señalar que la herramienta permite generar los componentes utilizando distintas plantillas y que la generación en este caso de estudio sólo obedece a la interpretación particular de la línea de producción que se seleccionó.

La proceso de generación del código Back-End, es el *paso 2 – Generación de código Back-End* de la Figura 55.

7.1. Lenguaje de definición de datos (DDL)

Una de las bondades en la carga de una definición de entidad, es que se permite configurar la sintaxis de carga con expresiones regulares. De este modo, podemos cargar definiciones de datos para distintas sintaxis de definición de datos, ver Figura 63. Sin embargo, alternativamente podemos integrar los datos de las entidades de forma manual.

```
1.  CREATE TABLE VIR010_AULA (
2.      CD_INSTITUCION      INTEGER NOT NULL,
3.      CD_AULA             CHAR(6) NOT NULL,
4.      TP_ACCESO           CHAR(2) NOT NULL,
5.      TP_CALENDARIO       CHAR(2) NOT NULL,
6.      ST_AULA             CHAR(2) NOT NULL,
7.      CD_INSTRUCTOR       INTEGER NOT NULL,
8.      CD_ACTIVACION       CHAR(10) NOT NULL,
9.      CD_ASIGNATURA       CHAR(4) NOT NULL,
10.     FH_MODIFICO         DATE NOT NULL,
11.     NB_AULA             VARCHAR(60) NOT NULL,
12.     TX_COMMENT          VARCHAR(256) NOT NULL,
13.     PRIMARY KEY (CD_INSTITUCION, CD_AULA)
14. );
```

Figura 63. Especificación del Lenguaje de Definición de Datos (DDL).

7.2. La generación de *Beans* para el mapeo de datos

La generación de un componente tipo *bean,* se logra utilizando la especificación de una entidad de datos. El proceso de generación consulta las propiedades que forman parte de la entidad y genera una clase con una interfaz que corresponden a cada campo de la entidad origen.

7.3. Especificación del servicio XML

Cuando el usuario inicia la generación de código *Back-End*, la herramienta arma un *stream* XML con la información del sistema. Esta información se extrae desde la base de datos de la herramienta y se representa ordenadamente para que posteriormente sea manipulada por el intérprete de *LENCOS*.

Ejemplo 1: Servicio XML para el paso 2 – Generación del Back-End.

```
1.  <clase>
2.      <nbclase>Aula</nbclase>
3.      <archclase></archclase>
4.      <commclase>Clase que tiene la información relativa al aula virtual</commclase>
5.      <nbentidadato></nbentidadato>
6.      <llavecompuesta>No</llavecompuesta>
7.      <clavariables>
8.          <variable>
9.              <nombre>scdinstitucion</nombre>
10.             <tipo>String</tipo>
11.             <valor></valor>
12.             <llave>Si</llave>
13.             <dbcampo>CD_INSTITUCION</dbcampo>
14.             <dblongitud>0</dblongitud>
15.             <dbdecimales>0</dbdecimales>
16.             <formatear>No</formatear>
17.             <validanull>No</validanull>
18.             <parametro>pscd</parametro>
19.             <comentario>Clave de la institución a la que pertenece el portal</comentario>
20.         </variable>
21.         <variable>
22.             <nombre>scdaula</nombre>
23.             <tipo>String</tipo>
```

```
            <valor></valor>
            <llave>Si</llave>
            <dbcampo>CD_AULA</dbcampo>
            <dblongitud>0</dblongitud>
            <dbdecimales>0</dbdecimales>
            <formatear>No</formatear>
            <validanull>No</validanull>
            <parametro>pscd</parametro>
            <comentario>Clave del aula virtual</comentario>
        </variable>
    /*    - EL TAG <variable> SE REPITE PARA CADA UNO DE LOS CAMPOS */
    </clavariables>
    <clametodos></clametodos>
</clase>
<beans>
    <clase>
        <nombre>Aula</nombre>
        <selectone>Si</selectone>
         <selectall>Si</selectall>
        <selectcriterio>Si</selectcriterio>
        <selectsome>Si</selectsome>
        <update>Si</update>
        <insert>Si</insert>
        <delobj>Si</delobj>
        <delkey>Si</delkey>
        <tabla>
            <nombre>VIR010_AULA</nombre>
            <campos>
                <campo>
                    <variable>scdinstitucion</variable>
                    <comentario>Clave de la institución en el portal</comentario>
                    <nombre>CD_INSTITUCION</nombre>
                    <tipo>String</tipo>
                    <llave>Si</llave>
                    <autoincremento>No</autoincremento>
                    <esecuencia>No</esecuencia>
                    <nbsecuencia></nbsecuencia>
                    <dblong>3</dblong>
                    <dbdecimales>0</dbdecimales>
                    <propiedad>cdinstitucion</propiedad>
                    <metodoget>getcdinstitucion</metodoget>
                    <metodoset>setcdinstitucion</metodoset>
                </campo>
                <campo>
                     <variable>scdaula</variable>
```

```
                    <comentario>Clave del aula virtual</comentario>
                    <nombre>CD_AULA</nombre>
                    <tipo>String</tipo>
                    <llave>Si</llave>
                    <autoincremento>Si</autoincremento>
                    <esecuencia>No</esecuencia>
                    <nbsecuencia></nbsecuencia>
                    <dblong>6</dblong>
                    <dbdecimales>0</dbdecimales>
                    <propiedad>cdaula</propiedad>
                    <metodoget>getcdaula</metodoget>
                    <metodoset>setcdaula</metodoset>
                </campo>
            </campos>
        </tabla>
    </clase>
</beans>
```

7.4. Meta-Lenguaje

Los componentes de la línea de producción que se van a tratar en este caso de estudio, son el bean y el componente de acceso a datos. El contenido del *script* en meta-lenguaje se puede ver en el siguiente ejemplo.

Ejemplo 2: Meta-lenguaje para generación de una clase tipo bean

```
<?php
/**
* @name: #=clase.nbclase#
* @creation date: (#=general.fhcreacion#)
* Descripción:
* #=clase.commclase#
* @author: #=general.nbprogramador#
* @company: #=proyecto.nbempresa#
**/

class #=clase.nbclase# {
    #PARA CADA variable EN clase.clavariables#
    #SI variable.tipo ESIGUALA String#
```

```
14.       var $#=variable.nombre# = '';              // #=variable.comentario#
15.       #FIN SI#
16.       #SI variable.tipo ESIGUALA boolean#
17.       var $#=variable.nombre# = false;          // #=variable.comentario#
18.       #FIN SI#
19.       #SI variable.tipo ESIGUALA int#
20.       var $#=variable.nombre# = 0;              // #=variable.comentario#
21.       #FIN SI#
22.       #SI variable.tipo ESIGUALA Integer#
23.       var $#=variable.nombre# = 0;              // #=variable.comentario#
24.      #FIN SI#
25.      #SI variable.tipo ESIGUALA double#
26.      var $#=variable.nombre# = 0;              // #=variable.comentario#
27.      #FIN SI#
28.      #SI variable.tipo ESIGUALA long#
29.      var $#=variable.nombre# = 0;              // #=variable.comentario#
30.      #FIN SI#
31.      #FIN PARA CADA#
32.
33.   /**
34.   * CONSTRUCTOR
35.   */
36.   function #=clase.nbclase#() {
37.   }
38.
39.   #PARA CADA variable EN clase.clavariables#
40.   /**
41.   *  PROPIEDAD: #=variable.comentario#
42.   *  @return valor de la propiedad
43.   */
44.   function get#=substring(variable.nombre, 1)#() {
45.   return $this->#=variable.nombre#;
46.   }
47.
48.   /**
49.   *  PROPIEDAD: #=variable.comentario#
50.   *  @param $val - nuevo valor
51.   */
52.   function set#=substring(variable.nombre, 1)#($val) {
53.   $this->#=variable.nombre# = $val;
54.   }
55.   #FIN PARA CADA#
56.
57.   /**
58.   *  PROCEDIMIENTO para determina si los objetos tienen la misma llave
```

```
59.  *  @param $obj - objeto
60.  */
61.  function hasamekey($obj) {
62.  # PARA CADA variable EN clase.clavariables DONDE llave ESIGUALA Si #
63.  if($this->get#=substring(variable.nombre, 1)#()==$obj->get#=substring(variable.nombre,
       1)#())
64.  #FIN PARA CADA#
65.  return true;
66.  return false;
67.  }
68.
69.  } /* fin clase [#=clase.nbclase#] */
70.  ?>
```

En el Ejemplo 2, se detalla el código que participa en la generación de un componente tipo *bean*. El código está integrado por una parte estática y dinámica, la parte estática representa la sintaxis del lenguaje de programación, y la parte dinámica esta formada por los textos de la forma *# instrucción #*, donde *instrucción* es una operación o palabra clave de *LENCOS*.

El Ejemplo 2 tiene la información general de la empresa y el autor entre las líneas 3 y 8. A partir de la línea 12 y hasta la 31, se concreta en recorrer cada una de las variables con el objetivo de declarar e inicializar las propiedades de la clase, la inicialización de valores depende del tipo de dato con el cual estamos tratando. Todo el código está debidamente documentado, porque se imprimen los comentarios de cada entidad y campo que llegaron a través del *XML*.

Entre la línea 39 y 55 del Ejemplo 2, se vuelven a recorrer las variables para crear las interfaces del objeto *get()* y *set()*, estas interfaces permiten asignar o leer el valor de las propiedades de la clase.

Ejemplo 3: Meta-lenguaje para la generación de acceso a datos

```
1.      <?php
2.
3.       require_once("../comun/Criterio.php");
4.       require_once("../util/Utilerias.php");
5.       require_once("../util/DBConnect.php");
6.
7.      /**
8.       * @name: #=proyecto.dbclase.nombre#
9.       * Descripcion:
10.      * #=proyecto.dbclase.comentario#
11.      * @creation date: (#=general.fhcreacion#)
12.      * @author #=general.nbprogramador#
13.      * @conpany: #=proyecto.nbempresa#
14.      */
15.     class #=proyecto.dbclase.nombre# {
16.       var $resultado = null;
17.       var $odatabase = null;
18.
19.       /**
20.        * CONSTRUCTOR
21.        */
22.       function #=proyecto.dbclase.nombre#() {
23.         $this->odatabase = new DBConnect();
24.         $this->odatabase->connect();
25.       }
26.
27.       /**
28.        *  FUNCION: Para referenciar el objeto dbconnect
29.        */
30.       function getdatabase() {
31.          return $this->odatabase;
32.       }
33.
34.       ////////////////////////////////////////////////////////////////////////
35.       //                        CODIGO SELECT
36.       ////////////////////////////////////////////////////////////////////////
37.
38.       #PARA CADA clase EN beans#
39.
40.       #SI clase.selectone ESIGUALA Si #
41.       /**
42.        *  PROCEDIMIENTO: Procedimiento para seleccionar un registro de la tabla:
```

```
43.      *  #=aminuscula(clase.tabla.nombre)#
44.      #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si #
45.      *  @param $p#=campo.variable# - #=campo.comentario#
46.      #FIN PARA CADA#
47.      *  @return $resultado
48.      */
49.     function get#=clase.nombre#(#_#
50.        #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si#
51.           #SI campo.esultimo ESIGUALA Si#
52.              $p#=campo.variable#) {
53.           #Y SI NO#
54.              $p#=campo.variable#, #_#
55.           #FIN SI#
56.        #FIN PARA CADA#
57.        $sql = 'SELECT #_#
58.       #PARA CADA campo EN clase.tabla.campos#
59.         #=aminuscula(campo.nombre)##_#
60.         #SI campo.esultimo ESIGUALA Si#
61.         ';
62.        #Y SI NO#
63.         , #_#
64.        #FIN SI#
65.      #FIN PARA CADA#
66.      $sql.= ' FROM  #=clase.tabla.nombre#';
67.      $sql.= " WHERE #_#
68.      #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si#
69.         #SI campo.esprimero ESIGUALA No#
70.           $sql.= " AND   #_#
71.         #FIN SI#
72.         #SI campo.tipo ESIGUALA String#
73.        #=aminuscula(campo.nombre)#='".$p#=campo.variable#."'";
74.       #Y SI NO#
75.        #=aminuscula(campo.nombre)#=".$p#=campo.variable#."";
76.        #FIN SI#
77.      #FIN PARA CADA#
78.       $resultado = $this->odatabase->select($sql);
79.     return $resultado;
80.    }
81.
82.   #FIN SI#
83.
84.  #SI clase.selectcriterio ESIGUALA Si#
85.    /**
86.     * PROCEDIMIENTO : consulta de algunos los registros de la tabla:
87.     *   #=aminuscula(clase.tabla.nombre)# en base a un criterio
```

```
88.         * @param  $ocrit - criterios de consulta
89.         * @return $resultado
90.         */
91.         function getc#=clase.nombre#s($ocrit) {
92.           $outil = new Utilerias();
93.           $swhere ='';
94.           $sql = 'SELECT #_#
95.           #PARA CADA campo EN clase.tabla.campos#
96.           #=aminuscula(campo.nombre)##_#
97.           # SI campo.esultimo ESIGUALA Si #
98.             ';
99.           #Y SI NO#
100.            , #_#
101.          #FIN SI#
102.        #FIN PARA CADA#
103.        $sql.= ' FROM  #=clase.tabla.nombre#';
104.        /*
105.        if($ocrit->hasinstitucion())
106.          $swhere = $outil->addToken($swhere,
107.                    "cd_institucion='".$ocrit->getcdinstitucion()."'",
108.                     " AND ");
109.        if($ocrit->hasplantel())
110.          $swhere = $outil->addToken($swhere,
111.                    "cd_plantel='".$ocrit->getcdplantel()."'",
112.                     " AND ");
113.        */
114.        if($swhere!='')
115.          $sql.= ' WHERE '.$swhere;
116.        $resultado = $this->odatabase->select($sql);
117.        return $resultado;
118.      }
119.    #FIN SI#
120.    #FIN PARA CADA#
121.
122.     /**
123.      *  PROCEDIMIENTO: Procedimiento para contar los registros de una tabla con su
124.                                  condición
125.      *  @param  $tabla - Nombre de la tabla
126.      *  @param  $swhere - Condición de la tabla
127.      *  @return $resultado
128.      */
129.     function getallCount($tabla, $swhere) {
130.        $sql = 'SELECT COUNT(*) AS num';
131.        $sql.= ' FROM '.$tabla;
132.        if($swhere!='')
```

```
133.       $sql.= ' WHERE '.$swhere;
134.     $resultado = $this->odatabase->select($sql);
135.     return $resultado;
136.   }
137.
138. ////////////////////////////////////////////////////////////////////////
139. //                        CONSECUTIVOS
140. ////////////////////////////////////////////////////////////////////////
141. #PARA CADA clase EN beans#
142. #PARA CADA campo EN clase.tabla.campos DONDE campo.autoincremento
       ESIGUALA Si#
143.   /**
144.    *  PROCEDIMIENTO: Procedimiento para consultar el ultimo id de la tabla
145.    *  #=aminuscula(clase.tabla.nombre)#
146.    # PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si #
147.    *  @param $p#=campo.variable# - #=campo.comentario#
148.    #FIN PARA CADA#
149.    *  @return $resultado
150.    */
151.   function getlast#=clase.nombre#(#_#
152.         #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si
        #
153.      #SI campo.esultimo ESIGUALA Si #
154.      $p#=campo.variable#) {
155.      #Y SI NO#
156.      $p#=campo.variable#, #_#
157.      #FIN SI#
158. #FIN PARA CADA#
159. $sql = 'SELECT MAX(#=aminuscula(campo.nombre)#)';
160.     $sql.= ' FROM  #=clase.tabla.nombre#';
161.     $sql.= " WHERE #_#
162.     #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si #
163.       #SI campo.esprimero ESIGUALA No #
164.     $sql.= " AND   #_#
165.       #FIN SI#
166.       #SI campo.tipo ESIGUALA String #
167.         #=aminuscula(campo.nombre)#='".$p#=campo.variable#."'";
168.       #Y SI NO#
169.         #=aminuscula(campo.nombre)#=".$p#=campo.variable#."";
170.       #FIN SI#
171.     #FIN PARA CADA#
172.     $resultado = $this->odatabase->select($sql);
173.     return $resultado;
174.   }
175.   #FIN PARA CADA#
```

```
176.    #FIN PARA CADA#
177.
178.    ////////////////////////////////////////////////////////////////////////
179.    //                      CODIGO INSERT
180.    ////////////////////////////////////////////////////////////////////////
181.    #PARA CADA clase EN beans #
182.    #SI clase.insert ESIGUALA Si #
183.    /**
184.     * PROCEDIMIENTO : inserta un registro nuevo en tabla
185.                                  #=aminuscula(clase.tabla.nombre)#
186.     * @param  $oadd - objeto tipo #=clase.nombre# a insertar
187.     * @return true / false
188.     */
189.    function add#=clase.nombre#($oadd) {
190.        $sql = 'INSERT INTO #=clase.tabla.nombre#';
191.        $sql.= '(#_#
192.        #PARA CADA campo EN clase.tabla.campos#
193.          #=aminuscula(campo.nombre)##_#
194.          #SI campo.esultimo ESIGUALA No #
195.            , #_#
196.          #FIN SI#
197.        #FIN PARA CADA#
198.        )';
199.        $sql.= " VALUES (#_#
200.        #PARA CADA campo EN clase.tabla.campos#
201.          #SI campo.tipo ESIGUALA String#
202.                  '".$oadd->#=campo.metodoget#()."'#_#
203.               #Y SI NO#
204.                  ".$oadd->#=campo.metodoget#()."#_#
205.               #FIN SI#
206.               #SI campo.esultimo ESIGUALA Si #
207.               )";
208.               #Y SI NO#
209.               ,#_#
210.               #FIN SI#
211.            #FIN PARA CADA#
212.        $resultado = $this->odatabase->select($sql);
213.        return $resultado;
214.      }
215.    #FIN SI#
216.    #FIN PARA CADA#
217.    ////////////////////////////////////////////////////////////////////////
218.    //                      CODIGO UPDATE
219.    ////////////////////////////////////////////////////////////////////////
220.    #PARA CADA clase EN beans#
```

```
#SI clase.update ESIGUALA Si #
 /**
  * PROCEDIMIENTO : actualización de un registro de la tabla
                                #=aminuscula(clase.tabla.nombre)#
  * @param  $oupd - objeto tipo #=clase.nombre# a actualizar
  * @return true / false
  */
 function upd#=clase.nombre#($oupd) {
   $sql = 'UPDATE #=clase.tabla.nombre#';
   $sql.= " SET #_#
  #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA No #
    #SI campo.esprimero ESIGUALA No #
    $sql.= ",#_#
    #FIN SI#
    #SI campo.tipo ESIGUALA String#
       #=aminuscula(campo.nombre)#= '".$oupd->#=campo.metodoget#()."'";
    #Y SI NO#
       #=aminuscula(campo.nombre)#= ".$oupd->#=campo.metodoget#()."";
    #FIN SI#
  #FIN PARA CADA#
  $sql.= " WHERE #_#
  #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si #
    #SI campo.esprimero ESIGUALA No #
  $sql.= " AND   #_#
    #FIN SI#
    #SI campo.tipo ESIGUALA String #
      #=aminuscula(campo.nombre)#='".$oupd->#=campo.metodoget#()."'";
    #Y SI NO#
      #=aminuscula(campo.nombre)#=".$oupd->#=campo.metodoget#()."";
    #FIN SI#
  #FIN PARA CADA#
  $resultado = $this->odatabase->select($sql);
  return $resultado;
 }
#FIN SI#
#FIN PARA CADA#
////////////////////////////////////////////////////////////////////////////
//                      CODIGO DELETE
////////////////////////////////////////////////////////////////////////////
#PARA CADA clase EN beans #
#SI clase.delobj ESIGUALA Si #
/**
 * PROCEDIMIENTO : elimina un registro de la tabla
                                #=aminuscula(clase.tabla.nombre)#
 * @param  $odel - objeto tipo #=clase.nombre# a eliminar
```

```
266.      * @return true / false
267.     */
268.     function del#=clase.nombre#($odel) {
269.       $sql = 'DELETE FROM #=clase.tabla.nombre#';
270.       $sql.= " WHERE #_#
271.       #PARA CADA campo EN clase.tabla.campos DONDE campo.llave ESIGUALA Si #
272.        #SI campo.esprimero ESIGUALA No#
273.       $sql.= " AND   #_#
274.        #FIN SI#
275.        #SI campo.tipo ESIGUALA String#
276.          #=aminuscula(campo.nombre)#="'.$odel->#=campo.metodoget#().'"';
277.        #Y SI NO#
278.          #=aminuscula(campo.nombre)#=".$odel->#=campo.metodoget#().'"';
279.        #FIN SI#
280.       #FIN PARA CADA#
281.       $resultado = $this->odatabase->select($sql);
282.       return $resultado;
283.     }
284.     #FIN SI#
285.     #FIN PARA CADA#
286.     } /* fin clase [#=proyecto.dbclase.nombre#]  */
287.     ?>
```

El Ejemplo 3, es el *script* para crear un objeto dedicado en exclusiva al acceso a la base de datos. El mecanismo de conexión a la base de datos se hace a través de las funciones propias de *PHP* y los datos se consultan o actualizan armando directamente la cadena con la instrucción *SQL* que sea requerida.

Entre las líneas 3 y 5 del Ejemplo 3, se puede identificar el uso de tres librerías que también serán integradas en el procedimiento de generación de código, y que forman parte de los componentes de la arquitectura. También es importante ver que en la línea 38, se recorren las clases que fueron construidas en el mismo proceso. Al finalizar la interpretación del *script*, se tendrá una clase con las funciones necesarias para el acceso a los datos.

En el Ejemplo 3, se está incorporando una función que es utilizada como utilería del producto final, muchas de estas definiciones son resultado de afinaciones sobre la línea de producción.

7.5. Componente resultante

El resultado del proceso de generación *Back-End* se puede visualizar en el Ejemplo 4. El resultado ya no tiene ninguna palabra reservada del meta-lenguaje, ni siquiera alguna sección entre los caracteres # y #, ese resultado da cuenta que el *script* ha sido procesado exitosamente.

Ejemplo 4: Componente resultante tipo bean.

```
1.  <?php
2.
3.  /**
4.   * @name: Aula
5.   * @creation date: (30-Nov-2011)
6.   * Descripción:
7.   * Clase que tiene la información relativa a cada aula virtual
8.   * @author: ...
9.   * @conpany: ...
10.  **/
11.
12.   class Aula {
13.       var $scdinstitucion = '';          // Clave de la institución a la que pertenece el portal
14.       var $scdaula = '';                 // Clave del aula virtual
15.
16.     /**
17.       * CONSTRUCTOR
18.     */
19.     function Aula() {   }
20.
21.     /**
22.       *  PROPIEDAD: Clave de la institución a la que pertenece el portal
23.       *  @return valor de la propiedad
24.       */
25.     function getcdinstitucion() {
26.         return $this->scdinstitucion;
27.      }
28.
29.      /**
30.       *  PROPIEDAD: Clave de la institución a la que pertenece el portal
31.       *  @param $val - nuevo valor
32.       */
33.      function setcdinstitucion($val) {
```

```
34.          $this->scdinstitucion = $val;
35.      }
36.
37.      /**
38.       *  PROPIEDAD: Clave del aula virtual
39.       *  @return valor de la propiedad
40.       */
41.      function getcdaula() {
42.          return $this->scdaula;
43.       }
44.
45.      /**
46.       *  PROPIEDAD: Clave del aula virtual
47.       *  @param $val - nuevo valor
48.       */
49.      function setcdaula($val) {
50.          $this->scdaula = $val;
51.       }
52.       /**
53.        *  PROCEDIMIENTO para determina si los objetos tienen la misma llave
54.        *  @param $obj - objeto
55.        */
56.       function hasamekey($obj) {
57.          if($this->getcdinstitucion()==$obj->getcdinstitucion())
58.          if($this->getcdaula()==$obj->getcdaula())
59.           return true;
60.        return false;
61.     }
62.    } /* fin clase [Aula] */
63.    ?>
```

8. GENERACIÓN DEL FRONT END

El proceso de integración se debe realizar después de haber terminado el código *Back-End* (*ver método en capítulo III*). La integración tiene como insumos el prototipo, la información de pantallas y el código *Back-End*, este proceso tiene una interfaz de usuario que se describe en el *Apéndice D - Sección 6*.

Internamente el proceso genera una especificación *XML* que suministra todas las características de la pantalla y que se muestra a detalle en el Ejemplo 5.

La proceso de generación del código Front-End, es el *paso 5 – Generación de código Front-End* de la Figura 55.

Ejemplo 5: Servicio XML para generación del Front-End

```
1.  <general>
2.  <nbcomponente>addAula</nbcomponente>
3.  <extcgi>.php</extcgi>
4.  <cdproyecto>000005</cdproyecto>
5.  <nbproyecto>CENESVIR</nbproyecto>
6.  <nbempresa>HANYGEN SOFTWARE S.A. DE C.V.</nbempresa>
7.  <nbcliente>CENTRO DE LENGUAS EXTRANJERAS</nbcliente>
8.  </general>
9.  <pantalla>
10. <cdproyecto>000005</cdproyecto>
11. <cdpantalla>000001</cdpantalla>
12. <nbimagefile>C:\Users\RUBEN\Google Drive\tesis
                    final\prototipo\images\addaula.jpg</nbimagefile>
13. <txcomment>Comentario por default</txcomment>
14. <txobjetivo>Pantalla para dar de alta una nueva aula virtual</txobjetivo>
15. <cdanalista>000000</cdanalista>
16. <nbanalista></nbanalista>
17. <cdprogramador>000000</cdprogramador>
18. <nbprogramador></nbprogramador>
19. <cdcodigo>PN000000</cdcodigo>
20. <stsincronia>SY</stsincronia>
21. <nbhtmlforma>frmaula</nbhtmlforma>
22. <nbcomponente>addAula</nbcomponente>
23. <nbhtmlfile>C:\Users\RUBEN\Google Drive\tesis
                    final\prototipo\addAula.html</nbhtmlfile>
24. <arreglos></arreglos>
25. <campos>
```

```
26.     <campo>
27.     <cdproyecto>000005</cdproyecto>
28.     <cdpantalla>000001</cdpantalla>
29.     <txcomment>Clave para ingresar nueva aula</txcomment>
30.     <cdcampo>000001</cdcampo>
31.     <tpcampo>000001</tpcampo>
32.     <tpinoutput>IN</tpinoutput>
33.     <nudecimales>0</nudecimales>
34.     <nulongitud>5</nulongitud>
35.     <cdarreglo></cdarreglo>
36.     <nbcampo>Clave</nbcampo>
```

El proceso de generación de código, manipula el código *HTML* para integrarlo con las interfaces de las clases, integra validaciones de captura y asocia cada campo con una interfaz. El resultado se puede ver en el Ejemplo 6.

Ejemplo 6: Componente resultante FRONT-END.

```
1.  <?php
2.  require_once("BIZAdmin.php");
3.  require_once("Aula.php");
4.  require_once("Asignatura.php");
5.  /**
6.    * Programa: addAula.php
7.    * Descripción:
8.    * Comentario por default
9.    * Autor:
10. **/
11. $odb = new DBConnect();
12. $odb->connect();
13. $conn = $odb->getconexion();
14. obusBIZAdmin = new BIZAdmin();
15. $mensaje = "";
16. iresultado = obusBIZAdmin->addAula(oadd);
17. ?>
18. <html>
19. <body>
20. <form name="frmaula" method="post">
21. <script language="javascript">
22. var form = document.frmaula;
23. var expregular = /^\d+$/
24. var expregular1 = /^\d+\.\d+$/
25. var expregular2 = /^\.\d+$/
```

```
26. var expemail = /^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@[a-zA-Z0-9-]{2,200}(\.[_a-zA-Z0-9-
                    ]+)+$/
27. function loading() {
28. <? if($mensaje!="") echo "alert(\".$mensaje.\");"; ?>
29. }
30. function valida_llave() {
31. return true;
32. }
33. function valida_datos() {
34. var stxtclave = delsqlchars(trim(form.txtclave.value));
35. var stxtnombre = delsqlchars(trim(form.txtnombre.value));
36. var scboinstructor = form.cboinstructor.value;
37. var scboasignatura = form.cboasignatura.value;
38. var stxtcdactivacion = delsqlchars(trim(form.txtcdactivacion.value));
39. var stxcomment = delsqlchars(trim(form.txcomment.value));
40. if(form.txcomment.value.length>256) {
41. alert('El campo [Comentario] excede las 256 posiciones permitidas');
42. return false;
43. }
44. return true;
45. }
46. function delsqlchars(s) {
47. var r = "";    /*del char ("), sencillo y doble*/
48. if(!s.match("[\",']")) return s;
49. for (i=0; i < s.length; i++) {
50. c = s.charAt(i);
51. if (c == '\"' || c == """) c = " ";
52. r += c;
53. }
54. return r;
55. }
56. function trim(cadena) {
57. var micadena = cadena;
58. while(micadena.charAt(0)==' ') micadena=micadena.substring(1,micadena.length);
59. while(micadena.charAt(micadena.length-1)==' ')
        micadena=micadena.substring(0,(micadena.length-1));
60. return micadena;
61. }
62. function salir() {
63. window.location.href = "../html/blanco.html";
64. }
65. </script>
66. <script language="javascript"></script>
67. <table>
68. <tbody>
```

```
69. <tr>
70. <td colspan="4"><b>DATOS DEL AULA VIRTUAL</b></td>
71. </tr>
72. <tr class="even">
73. <td width="5%"> </td>
74. <td>Clave: </td>
75. <td class="grey"><input type="text" name="txtclave" id="txtclave"  value="" class="input-
                short" disabled / maxlength5"> * autom&aacute;tica     </td>
76. </tr>
77. <tr class="even">
78. <td width="5%"> </td>
79. <td>Nombre: </td>
80. <td><input type="text" name="txtnombre" id="txtnombre"  value="" class="input-big"
                maxlength="60" /></td>
81. <td width="5%"> </td>
82. </tr>
83. <tr class="even">
84. <td width="5%"> </td>
85. <td>Profesor: </td>
86. <td>
87. <select name="cboinstructor" id="cboinstructor" class="small">
88. </select>
89. </td>
90. </tr>
91. <tr class="even">
92. <td width="5%"> </td>
93. <td>Asignatura: </td>
94. <td>
95. <select name="cboasignatura" id="cboasignatura" class="small">
96. </select>
97. </td>
98. </tr>
99. <tr class="even">
100. <td width="5%"> </td>
101. <td>C&oacute;digo de activaci&oacute;n: </td>
102. <td><input type="text" name="txtcdactivacion" id="txtcdactivacion"  value=" class="input-
            short"  maxlength="10" /></td>
103. </tr>
104. <tr class="even">
105. <td width="5%"> </td>
106. <td valign="top">Comentario: </td>
107. <td class="grey"><textarea name="txcomment" id="txcomment" class="input-big">
108. </textarea> max.256</td>
109. <td width="5%"> </td>
110. </tr>
```

```
111. </tbody>
112. </table>
113. <br />
114. <div align="center">
115. <input type="button" id="btnaddAula" name="btnaddAula" value=" Agregar "
        class="hanybutton" onclick="javascript:agregar();" />
116. <input type="button" id="btnClose" name="btnClose" value=" Cerrar "
        class="hanybutton" onclick="javascript:cerrar();" />
117. </div>
118. </form>
119. </body>
120. </html>
121. <?php
122. $odb->close_mysql();
123. ?>
```

CAPÍTULO V

CONCLUSIONES

Conclusiones

En la presente tesis se diseñó, implementó y validó experimentalmente una herramienta de soporte para el desarrollo semi-automatizado de sistemas de información. La herramienta aproxima la construcción de una nueva aplicación al generar una implementación "intermedia", la cual deberá ser refinada para obtener un sistema totalmente funcional que cumpla con todos los requerimientos del usuario.

El objetivo de la herramienta propuesta es soportar el establecimiento de una fábrica de software, que ayude a reducir los tiempos necesarios para el desarrollo de productos, y reduzca la posibilidad de fallas por medio de la reutilización de código altamente probado. Adicionalmente, con cada componente de software, también se genera su documentación correspondiente, lo que permite que un programador que retome el código pueda entender mejor su implementación.

La herramienta propuesta permite la incorporación de distintas tecnologías de implementación, y por ello, se creó un lenguaje de especificación con su correspondiente intérprete que es capaz de generar código en diferentes lenguajes de programación. El lenguaje propuesto, fue presentado por medio de su especificación en notación de Backus-Naur.

La interfaz gráfica se desarrolló de forma amigable con el usuario. La herramienta inicia con la solicitud de usuario y contraseña; la validación del usuario regresa los privilegios correspondientes en el menú principal. El menú principal organiza las funciones de acuerdo a su objetivo. Es así como encontramos el menú de *Documentación* que permite la captura de los datos y emisión de documentos, el menú de *Generación* que permite generar el código y el menú de *Herramientas* que integra las facilidades adicionales para el usuario.

La posibilidad de incluir diferentes arquitecturas de software, y de diferentes lenguajes de programación, permite a los arquitectos de una organización, reconfigurar la herramienta propuesta para otras líneas de producción. La capacidad de reconfiguración, da pie a las organizaciones a cumplir con estándares de calidad corporativos en el desarrollo de sistemas. Normalmente estas reglas de codificación suelen cambiar con la generación de nuevas versiones y en eso, la herramienta también permite ser reconfigurada y adaptada.

Las reglas de calidad en la codificación, son especificaciones muy puntuales de cómo debe estar escrito el código de los programas. Su definición persigue objetivos de

claridad y optimización del código. A la fecha, las grandes corporaciones, integran programas validadores, para que estas reglas sean consideradas cabalmente.

Es importante mencionar, que la participación de los arquitectos de software se limita a la configuración de las líneas de productos, estas configuraciones se ponen a punto con el primer producto generado y después de tener una configuración estable y optimizada, la única participación del arquitecto es de asesorías esporádicas.

Tomando en cuenta que tenemos un ahorro en los recursos humanos, además de tener un producto rápido, tendremos gastos de producción menores en la generación de sistemas informáticos y una optimización del capital humano con conocimientos altamente especializados.

La optimización de capital humano también es considerada cuando tratamos con la documentación, ya que el recurso especializado disminuye sus labores de documentación del código existente, porque para ello, hay una función de la herramienta que la extrae de modo automático. Este proceso de extracción se logra utilizando la configuración de los arquitectos de software.

La herramienta fue probada en proyectos de software implementados en tecnologías de programación como Java, Visual Basic .Net, PHP y C#, comprobando así el objetivo que tratamos en esta investigación. También se lograron configurar diferentes estilos sobre un mismo lenguaje para satisfacer diferentes requerimientos de codificación.

La configuración actual de la herramienta está preparada para generar nuevas aplicaciones con una arquitectura CGI y con componentes codificados en Java, Visual Basic, PHP y C#. Adicionalmente, se tienen diferentes versiones en la implementación de componentes en Java y PHP; las versiones son producto de adaptaciones por estándares industriales para requerimientos de cliente específicos. Por otro lado, las plantillas (*templates*) de documentación son personalizadas por cada cliente que solicita una nueva aplicación.

De estas pruebas podemos concluir la efectividad de la plataforma de software para líneas de productos de software, programación generativa y su eficiencia en la implementación en la producción masiva. En todos los casos, la generación de código ayudó a tener sistemas de forma rápida y bien documentados. El tiempo de los expertos también fue optimizado para actividades especializadas. El grueso de las tareas repetitivas fueron automatizadas por la herramienta, de este modo, disminuyeron las actividades y duración del plan de trabajo.

Se comprobó que la herramienta ayudó a aproximar soluciones en mayor o menor medida, dependiendo de la especialización técnica o grado de reglas de negocio particulares.

2. APLICACIÓN EN PROYECTOS

Es importante mencionar que la herramienta se ha utilizado en distintos sectores de la economía, pero siempre en el desarrollo de sistemas de gestión o informáticos. Todos los proyectos variaron en la importancia y en el número de personas que trabajaron ellos; también se encontraron diferentes condiciones de premura, de capacidad del equipo de trabajo, estándares institucionales y tecnologías permitidas. En general, estas condiciones fueron superadas y la herramienta mostró su utilidad en los procesos de aproximación de sistemas.

Las tecnologías y estándares que se utilizaron dependían básicamente de las especificaciones solicitadas por cada cliente. En el mejor de los casos, se contaba con una configuración semejante dentro de la plataforma, por lo que se procedía a ajustarla para crear una nueva línea de producción ad-hoc al cliente. En algunos otros casos, se tenía que configurar completamente la arquitectura, los lenguajes, los estándares y las plantillas (*templates)* de la documentación.

La herramienta permite la configuración de diversas arquitecturas por medio de una configuración de la variabilidad; con lo que es posible generar aplicaciones en *frameworks* como *Execution Service*, *Struts*, *Spring* y *JSF*. La diversidad de los lenguajes de programación para generar el código de cada componente, se logra utilizando el lenguaje de programación de dominio específico. El lenguaje permite definir el código fijo y el código variable de cada componente para de este modo generar componentes en lenguajes como Visual Basic, Java, PHP, etc. En el contenido de cada componente, es posible encontrar código para el manejo de dispositivos o servicios externos, las particularidades requeridas también son logradas usando el lenguaje de dominio específico. Ver figura 64.

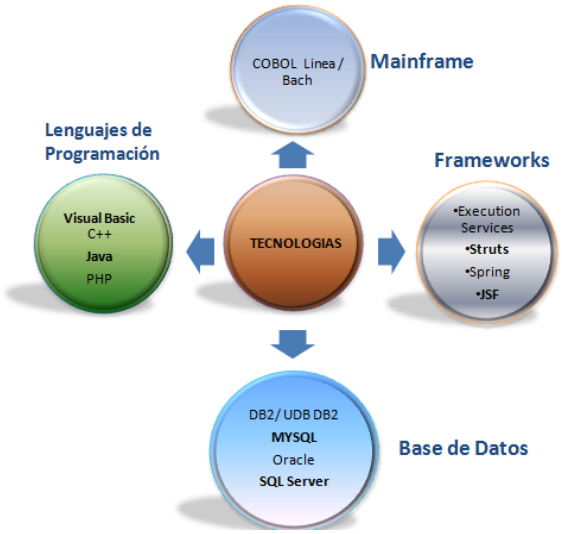

Figura 64. Tecnologías consideradas.

A continuación presentamos los proyectos en donde se utilizó la herramienta.

NAVISTAR MÉXICO

- **SISTEMA DE ADMINISTRACIÓN DE VENTAS**

  Sistema para administrar las ventas de camiones y refacciones a nivel nacional. El sistema se construyó en Visual Basic con el *framework* .NET y se ejecuta en el servidor IIS y participaron 4 personas.

BBVA BANCOMER

- **DESARROLLO DEL 90% DE LAS APLICACIONES DE INTRANET PARA EL ÁREA DE OPERACIONES, COMERCIAL Y RECURSOS MATERIALES EN AFORE**

  Sistemas para administrar las áreas comercial, operativa y recursos materiales con información de los clientes que están afiliados a la Afore. Los sistemas se construyeron en Java con el patrón MVC y se ejecuta en el servidor Websphere con una base de datos DB2 y un Mainframe IBM, participaron 18 personas.

- **ADMINISTRACIÓN DE FONDOS**

  Sistema para administrar a los trabajadores de fondos de previsión social en empresas que contratan a BBVA Bancomer. El sistema se construyó en Java con el patrón MVC y se ejecuta en el servidor Websphere con una base de datos DB2 y un Mainframe IBM, participaron 18 personas.

ADQUIRA MOVISTAR

- **TERMINALES DE COBRO EN PYMES**

  Sistemas para realizar cobro en terminales inteligentes. El sistema *Stand Alone*, se construyó en Visual Basic con el patrón MVC y una base de datos en SQL Lite. Participaron 2 personas.

- **HERRAMIENTA DE HOMOLOGACIÓN DE FORMATOS**

  Sistemas para homologar formato de los archivos operativos de transacciones. El sistema *Stand Alone*, se construyó en Visual Basic con el patrón MVC y una base de datos en SQL Lite. Participaron 2 personas.

HSBC

- **BUSINESS CONTINUITY PLAN (BCP)**

  Sistemas para administrar los procesos de contingencia de instalaciones y sistemas bancarios. El sistema se construyó en Java con un *framework* llamado Execution Service y se ejecuta en el servidor Websphere con una base de datos DB2 y un Mainframe IBM, participaron 8 personas.

- **ADMINISTRACIÓN DE PROVEEDORES (LEAP)**

  Sistemas para administrar los proveedores del banco. El sistema se construyó en Java con un *framework* llamado Execution Service y una base de datos en ORACLE.

- **CUSTOMER RELATIONSHIP MANAGEMENT (CRM)**

  Sistemas para administración de ventas. Se construyó en Java con un *framework* llamado Execution Service y se ejecuta en el servidor Websphere con una base de datos DB2 y un Mainframe IBM.

- **SISTEMA DE CALIFICACIÓN (SICAL)**

  Migración del sistema de calificación de clientes. El sistema se construyó en Java con un *framework* llamado Execution Service y una base de datos en ORACLE.

SEP

- **SISTEMA DE ADMINISTRACIÓN PRESUPUESTAL**

  Sistema para administrar la programación presupuestal de las áreas y departamentos. El sistema se construyó en Visual Basic con el *framework* .NET y se ejecuta en el servidor IIS y participaron 3 personas.

INFONAVIT

- **ADMINISTRACIÓN DE APORTACIONES IMSS**

  Sistema para administrar las aportaciones que llegan a través del IMSS. El sistema se construyó en Java con el *framework* STRUTS y se ejecuta en el servidor Websphere y participaron 3 personas.

CLARO PUERTO RICO

- **SISTEMA DE CONTROL VEHICULAR**

  Sistema FAW para administrar el mantenimiento mecánico de las flotillas vehiculares. El sistema se construyó en Java con el *framework* STRUTS y se ejecuta en el servidor Websphere, participaron 5 personas.

HANYGEN SOFTWARE

- **CENTRO ESCOLAR VIRTUAL (CENESVIR)**

  Sistema para administrar servicios a instituciones educativas. El sistema se construyó en PHP con un modelo MVC y una base de datos MySQL.

- **MULTI-AGENDA WEB**

  Sistema para administrar actividades. El sistema se construyó en PHP con un modelo MVC y una base de datos MySQL.

FARMACIAS SIMILARES

- **ADMINISTRACIÓN DE CONSULTAS MEDICAS**

  Sistema para administrar las consultas médicas con un control detallado de las recetas, enfermedades y clientes. El sistema se construyó en Visual Basic con el *framework* .NET y se ejecuta en el servidor IIS y participaron 3 personas.

- **PORTAL MÉXICO Y LATINO AMÉRICA**

  Sistema para el control de ventas en sucursales y franquicias de México y Latino América. El sistema se construyó en Visual Basic con el *framework* .NET y se ejecuta en el servidor IIS, y participaron 6 personas.

3. TRABAJOS FUTUROS

Hasta ahora, la investigación se ha centrado en diseñar una herramienta que permita definir una línea de producción de software. Sin embargo, el área de ingeniería de software con sus tecnologías, seguirá innovando en las diferentes vertientes de metodologías, administración, lenguajes, dispositivos electrónicos, etc. Cada avance en las diversas áreas de la ingeniería de software, permitirá profundizar o ampliar el paradigma de línea de producto de software.

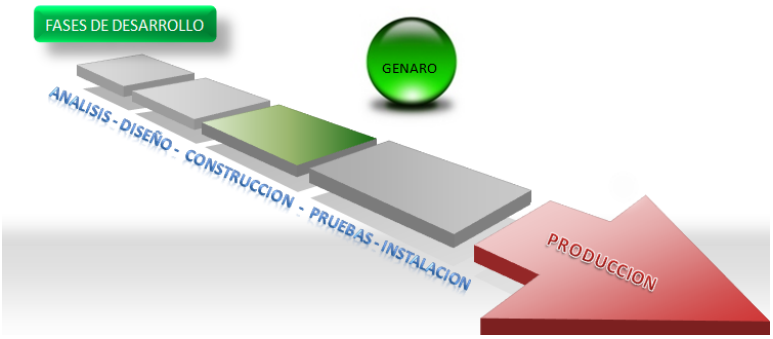

Figura 65. Influencia de plataforma en las fases de desarrollo de un sistema.

La plataforma para la línea de productos de software que se implementó sólo considera los productos de la fase de construcción (Ver Figura 65). La herramienta no califica, no audita y tampoco tiene funciones para ninguna área de gestión de proyectos. Sin embargo, se tiene información relevante para otras fases del proceso de desarrollo, como puede ser: la puesta a punto y mantenimiento del código, también se tiene información para otras áreas interesadas en productividad y calidad.

La metodología, sólo considera la interacción con programadores y documentadores que producen directamente los tangibles, pero deja a un lado al personal de gestión que también interviene en el éxito del proceso. Un diseño de metodología que incluya la iteración con áreas administrativas, necesariamente implica la creación de nuevas funciones en la plataforma tecnológica.

En el proceso de investigación se han identificado tres aspectos que podrían ser un marco a futuras investigaciones, como lo muestran los planos de la Figura 66. Estos tres aspectos fueron abordados de forma superficial, sin embargo, lo suficiente para imaginar los temas que podrían surgir y complementar esta investigación.

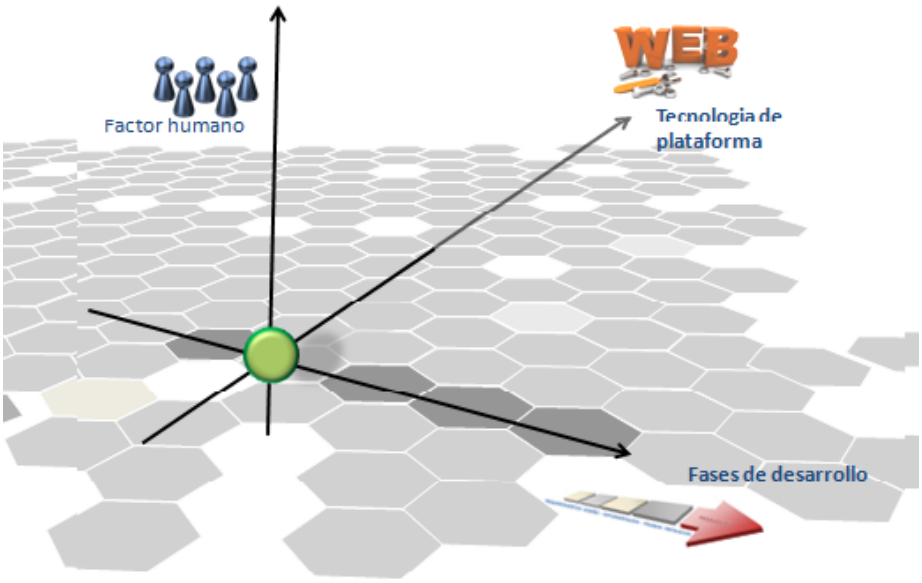

Figura 66. Aspectos generales de futuras investigaciones.

Las investigaciones relacionadas con el factor humano, podrían ser en el marco de la inteligencia artificial con o sin interacción a través de dispositivos electrónicos. De ahí que podría complementarse la etapa de levantamiento de requerimientos con una interfaz de comunicación por voz y su interpretación de lenguaje natural, o podría tratarse de una investigación exclusivamente en áreas de inteligencia artificial con el fin de crear personas virtuales, semejantes a los perfiles de trabajo en el proceso de análisis, diseño o administración a lo largo del proceso de creación de un software.

Las investigaciones respecto a la metodología y fases de desarrollo, podrían tener preponderancia administrativa con la intención de incluir cuestiones de mediciones, auditoría, calificación de riesgos, etc. Estos temas administrativos podrían interactuar con la herramienta o ser parte de ella, lo cual ampliaría el espectro de automatización del proceso de desarrollo. Hoy en día, hay fuerte interacción de las faces de desarrollo con el área de administración de proyectos, y es en esa área donde se buscaría el beneficio de futuros proyectos de investigación.

Es factible pensar en investigaciones sin necesidad de ampliar o abrirse hacia nuevas áreas, o dicho de otro modo, investigaciones que mejoren los diseños e implementaciones al interior de la herramienta. Estas nuevas investigaciones podrían incluir modelado y generación de métodos de negocio con el objetivo de lograr un porcentaje mayor en la generación de productos. También podrían incrementar la aplicabilidad del lenguaje de dominio específico a nuevas áreas u optimizar el algoritmo de interpretación.

Uno de los temas que motivaría a futuras investigaciones en el diseño de línea de producción, es la configuración de arquitecturas de software. En la investigación se consideraron aquellas que existen actualmente: MVC, Struts, Spring, Execution Service y JSF, sin embargo, estas arquitecturas se incrementan cada día y podrían dejar obsoleto al diseño actual.

Finalmente, el objetivo de la investigación fue lo suficientemente amplio, pero no se consideró incluir procesos con algoritmos optimizados. La investigación se concentró en un diseño comprobable de línea de producto de software que fuese implementado como herramienta. Esa es principalmente la razón, que motiva a pensar en mejorar el diseño de cada una de sus partes.

4. REFERENCIAS

[BARRY 2001] Barry Boehm. 2001. The Spiral Model as a Tool for Evolutionary Acquisition. pp.5

[CLEMENTS NORTHROP 2001] Clements, P. y Northrop, L. 2001. Software Product Lines: Practices and Patterns.

[COLBURN 2008] Alex Colburn, Jonathan Hsieh, Matthew Kehrt, Aaron Kimball. 2008. There is no Software Engineering Crisis

[OSCAR SALVA 2010] Oscar Diaz, Salva Trujillo. 2010. Fábricas de Software: experiencias, tecnologías y organización

[ELIZABETH KEN JEREMY 2011] Elizabeth Hull, Ken Jackson, Jeremy Dick.2011.Requirements Engineering.

[HARSU 2002] Maarit Harsu. 2002. A survey on domain engineering.pp.4

[HEYMANS TRIGAUX 2003] Patrick Heymans, Jean-Christophe Trigaux. 2003. PLENTY PROJECT - Software Product Lines: State of art

[HUNT 1999] Andrew Hunt, David Thomas. 1999. Pragmatic Programmer, pp 95-100

[IEEE 1990] IEEE.1990. Standard Glossary of Software Engineering Terminology IEEE S t (610).

[IEEE 1998] IEEE. 1998. Recommended Practice for Software Requirements Specifications

[KLAUS GUNTER FRANK 2005] Klaus Pohl, Günter Böckle, Frank van der Linden. 2005. Software Product Line Engineering - Foundations, Principles, and Techniques.

[FRANK KLAUS EELCO 2007] Frank van der Linden, Klaus Schmid, Eelco Rommes. 2007. Software Product Lines in Action

[KRZYSZTOF 1998] Krzysztof Czarnecki. 1998. Generative Programming

[KRUEGER CLEMENTS 2013] Charles W. Krueger, Paul Clements. 2013. Gears Product Line Engineering Tool and Lifecycle Framework - Product Overview

[LEWIS 1994] Lewis G. 1994. "What is Software Engineering?" DataPro (4015). pp. 1-10.

[MENDONCA BRANCO COWAN 2009] Marcilio Mendonca, Moises Branco, Donald Cowan. 2009. S.P.L.O.T.- Software Product Lines Online Tools

[NORTHROP 2002] Linda M. Northrop. 2002. SEI's Software Product Line Tenets

[OUALI KRAIEM GHEZALA 2012] Sami Ouali, Naoufel Kraiem, Henda Ben Ghezala. 2012. Intentional Software Product Line

[PAIS 2012] EL PAIS. 2012. EE UU tiene más desarrolladores de aplicaciones que agricultores.

[PARNAS 1976] David L.Parnas. 1976. IEEE Transactions on Software Engineering

[PRESSMAN 2001] Roger Pressman. 2005. Ingeniería de Software 5ta Edición. pp.4-7

[SEI 2013] Software Engineering Institute. 2013. Software Product Lines. Extraído el 25 de Mayo del 2013 desde fuente. http://www.sei.cmu.edu/productlines/

[SENNETT 2008] Richard Sennett. 2008. The Craftsman. pp.9

[SPLC 2013] Software Engineering Institute. 2013. Software Product Lines Conferences. Extraído el 22 de Mayo del 2013 desde fuente. http://www.splc.net/history.html

[WAYT 1994] W. WaytGibbs. 1994. SCIENTIFIC AMERICAN - Software's Chronic Crisis

[WIKI 2013] Wikipedia. 2013. Waterfall Model. Extraído el 21 de Agosto del 2013 desde la fuente. http://en.wikipedia.org/wiki/Waterfall_model

[ZAVE 1997] PAMELA ZAVE. 1997. Classification of Research Efforts in Requirements Engineering. pp.315

5. ACRÓNIMOS

BACK-END	Término del diseño de software que refiere a la parte que procesa los requerimiento de la interfaz del usuario.
CGI	Es un acrónimo de *Common Gateway Interface*.
DDL	*Definition Data Language* - Lenguaje proporcionado por el sistema de gestión de base de datos, que permite a los usuarios de la misma llevar a cabo las tareas de definición de las estructuras que almacenarán los datos, así como de los procedimientos o funciones que permitan consultarlos.
DOD	Es un acrónimo de *Department of Defense.* Es el ministerio del gobierno de Estados Unidos encargado de las fuerzas militares del país.
DSL	*Domain Specific Language* - Lenguaje que esté especializado en modelar o resolver un conjunto específico de problemas.
FRONT-END	Término del diseño de software que refiere a la parte que interactúa con el usuario.
GP	Es un acrónimo de *Generative Programming*.
IDE	Es un acrónimo de *Integrated Development Environment*
LENCOS	Es un acrónimo de *Lenguaje de Conversión de Servicios XML*
MVC	Es un acrónimo del *Model View Controller*, es un patrón o modelo de abstracción de desarrollo de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de negocio en tres componentes distintos.
RAD	*Rapid Application Development* – Un enfoque basado en el concepto de que los productos pueden ser desarrollados mas rápido y con mayor calidad.
SOA	*Service-oriented Arquitecture* – Conjunto de principios y metodología para el diseño y desarrollo de software en forma de servicios inter-operables.
SPL	Es un acrónimo del paradigma *Software Product Line.*
SQL	*Structured Query Language* - Lenguaje declarativo de acceso a bases

de datos relacionales que permite especificar diversos tipos de operaciones.

SRS *Software requirements specification*. es una descripción completa del comportamiento del sistema que se va a desarrollar. Incluye un conjunto de casos de uso que describe todas las interacciones que tendrán los usuarios con el software. Los casos de uso también son conocidos como requisitos funcionales.

STAND ALONE Software de computadora que puede trabajar fuera de línea, i.e. necesariamente no requiere una conexión de red para trabajar.

APÉNDICES

Apéndice A .

<instrucciones> ::= [<instrucción> | <línea-impresión>] {<instrucción> | <línea-impresión>}
<instrucción> ::= "#" [<impresión-valor> | <instrucción-ciclo> | <instrucción-si-condición> |
<instrucción-si-hay> | <instrucción-asignación> | <instrucción-incremento> |
<instrucción-usando>] "#"
<impresión-valor> ::= "=" [<expresión>]
<instrucción-ciclo> ::= "PARA CADA " <identificador> " EN " <contexto-id> {" DONDE "
<identificador> ["ESIGUALA" | "ESDIFERENTEA"] <constante>} [<instrucciones>]
"FIN PARA CADA"
< instrucción-si-condición> ::= "SI "<contexto-id> ["ESIGUALA"|"ESDIFERENTEA"|"ESTAEN"]
[<constante>| <identificador>] [<instrucciones>] "FIN SI"
< instrucción-si-hay> ::= "SI HAY "<identificador> " EN " <contexto-id> {" DONDE "
<identificador> ["ESIGUALA" | "ESDIFERENTEA"] <constante>} [<instrucciones>]
"FIN SI HAY"
< instrucción-asignación> ::= "ASIGNA" <constante> "AVARIABLE" <contexto-id>
< instrucción-incremento> ::= "INCREMENTA" <constante> "AVARIABLE" <contexto-id>
< instrucción-usando> ::= "USANDO" <identificador> "EN" < contexto-id> "POSICION"
<número> [<instrucciones>] "FIN USANDO"
< expresión> ::= [<contexto-id> | <función> "(" <contexto-id> { "," < número> }
{ "," < número> } ")"]
<contexto-id> ::= <contexto>{.<contexto>}.<identificador>
<contexto> ::= {<identificador>}
<constante> ::= [<letra> | <digito>] {<letra> | <digito>}
<función> ::= [<letra>] { <letra> | <digito> | "_" | "-" }
<identificador> ::= [<letra>] { <letra> | <digito> | "_" | "-" }
<letra> ::= <minúscula> | <mayúscula>
<número> ::= <digito> | <número> <digito>
<minúscula> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |"j" | "k" | "l" | "m" | "n" | "˜n"
| "o" | p" | "q" |"r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<mayúscula> ::= "A" | "B" | "C" | "D" | "E" | "f" | "G" | "H" | "I" |"J" | "K" | "L" | "M" | "N"
| "˜N" | "O" "P" | "Q" |"R" | "S" | "T" | "U" | "V" | "W" | "X" | "y" | "Z"
<digito> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Apéndice B.

Configuración de expresiones regulares para lectura DDL

```
<!--
    Archivo de configuración para la lectura de un DDL
    a variables de un lenguaje en particular
-->
<estilo>
<general>
        <nombre>SQL Server 2000</nombre>
        <identificar_llaves>Si</identificar_llaves>
        <comment>Configuración de expresiones regulares para la lectura
            de un archivo de lenguaje de definición de datos estilo
            SQL Server 2000
        </comment>
        <autor></autor>
</general>
<exp-regulares>
        <expresion>
            <clave>01</clave>
            <nombre>tabla</nombre>
            <valor>(CREATE)\s+(TABLE)\s+(?<nombre>\S+)</valor>
        </expresion>
        <expresion>
            <clave>02</clave>
            <nombre>campo</nombre>
            <valor>(?<campo>\S+)\s+(?<tipo>\w+)(\(((?<long>\d+)|
            (?<long>\d+),\s*(?<dec>\d+))\))*\s*(?<null>,|NOT NULL|NULL),</valor>
        </expresion>
        <expresion>
            <clave>03</clave>
            <nombre>primarykey</nombre>
            <valor>(PRIMARY)\s+(KEY)\s+(NONCLUSTERED)*\s*\
                ((?<llave>(\S+\s*)+)\)</valor>
        </expresion>
        <expresion>
            <clave>04</clave>
            <nombre>campo</nombre>
```

```
            <valor>(?<campo>\S+)</valor>
        </expresion>
        <expresion>
            <clave>05</clave>
            <nombre>campos</nombre>
            <valor>(?<llave>\S+)</valor>
        </expresion>
        <expresion>
            <clave>06</clave>
            <nombre>nonull</nombre>
            <valor>NOT NULL</valor>
        </expresion>
        <expresion>
            <clave>07</clave>
            <nombre>reference</nombre>

        <valor>(ALTER)\s+(TABLE)\s+(?<tabla>\S+)\s+(ADD)\s*(FOREIGN)\s+(KEY)\s+\((?<pa
rams>([\w+,?\s*]+))\)\s+(REFERENCES)\s+(?<referencia>\w+)</valor>
        </expresion>

</exp-regulares>
</estilo>
```

Apéndice C.
Meta-arquitectura del Back-End

```
<code>
        <general>
                <lenguaje>PHP</lenguaje>
                <ide>Yahoo!</ide>
                <autor>…</autor>
                <fecha_creacion>15-Ene-2012</fecha_creacion>
                <wizard>wizard.xml</wizard>
                <comment>
                  Estilo para la generación de código Back-End en PHP para Yahoo.
                  Las características de su generación son:
                  [ ] 1. Genera los arreglos con tipo de dato array
                  [ ] 2. Esta versión aun no incluye el objeto Criterio
                  [ ] 3. La conección esta lista para MySQL usando mysql_connect()
                  [ ] 4. Las referencias a objetos se genera por default y debe ajustarse
                </comment>
        </general>
        <model>
                <codigo>
                        <clave>01</clave>
                        <nombre>property</nombre>
                        <descripcion>Meta-código de propiedades</descripcion>
                        <archivo>getset.met</archivo>
                        <extension>.php</extension>
                </codigo>
                <codigo>
                      <clave>02</clave>
                        <nombre>bean</nombre>
                        <descripcion>Meta-código para los beans de datos</descripcion>
                        <archivo>bean.met</archivo>
                        <extension>.php</extension>
                </codigo>
                <codigo>
                      <clave>03</clave>
                        <nombre>datos</nombre>
                        <descripcion>Meta código de clase de acceso a datos</descripcion>
                        <archivo>datos.met</archivo>
                        <extension>.php</extension>
                </codigo>
                <codigo>
```

```
            <clave>04</clave>
               <nombre>negocio</nombre>
               <descripcion>Meta-código de clase de negocio</descripcion>
               <archivo>negocio.met</archivo>
               <extension>.php</extension>
         </codigo>
</model>
<librerias>
         <libreria>
            <clave>01</clave>
               <nombre>dbconexion</nombre>
               <descripcion>Clase para conexión a base de datos</descripcion>
               <archivo>DBConnect.php</archivo>
               <clase>DBConnect</clase>
               <extension>.php</extension>
         </libreria>
</librerias>
<comentarios>
         <comentario>
                  <clave>01</clave>
                  <nombre>metodo</nombre>
                  <tipo>no_inicia_con</tipo>
                  <texto_inicio>@</texto_inicio>
         </comentario>
         <comentario>
                  <clave>02</clave>
                  <nombre>clase</nombre>
                  <tipo>no_inicia_con</tipo>
                  <texto_inicio>@</texto_inicio>
         </comentario>
         <comentario>
                  <clave>03</clave>
                  <nombre>pseudocodigo</nombre>
                  <tipo>inicia_con</tipo>
                  <texto_inicio>'\sPASO\sw+:</texto_inicio>
         </comentario>
</comentarios>
<parametros>
         <parametro>
                  <clave>01</clave>
                  <nombre>caracter_comentario_mismalinea</nombre>
                  <valor>//</valor>
         </parametro>
         <parametro>
                  <clave>02</clave>
```

```
        <nombre>chars_aeliminar_encomentario</nombre>
        <valor>*/</valor>
</parametro>
<parametro>
        <clave>03</clave>
        <nombre>extension_deprogramas</nombre>
        <valor>.php</valor>
</parametro>
<parametro>
        <clave>04</clave>
        <nombre>apuntador_clase</nombre>
        <valor>(class)\s(?<name>\w+)</valor>
</parametro>
<parametro>
        <clave>05</clave>
        <nombre>apuntador_comentario_clase</nombre>
        <valor>(/\*\*)\s*</valor>
</parametro>
<parametro>
        <clave>06</clave>
        <nombre>apuntador_comentario_metodo</nombre>
        <valor>(/\*\*)\s*</valor>
</parametro>
<parametro>
        <clave>07</clave>
        <nombre>apuntador_procedimiento</nombre>
        <valor>(function)\s+(?<name>\w+)</valor>
</parametro>
<parametro>
        <clave>08</clave>
        <nombre>apuntador_declaracion_variable</nombre>
        <valor>(var)\s+\$(?<nombre>\w+)</valor>
</parametro>
<parametro>
        <clave>09</clave>
        <nombre>directorio_beans</nombre>
        <valor>/beans</valor>
</parametro>
<parametro>
        <clave>10</clave>
        <nombre>directorio_datos</nombre>
        <valor>/datos</valor>
</parametro>
<parametro>
        <clave>11</clave>
```

```
                <nombre>directorio_negocio</nombre>
                <valor>/negocio</valor>
            </parametro>
            <parametro>
                <clave>12</clave>
                <nombre>directorio_librerias</nombre>
                <valor>/librerias</valor>
            </parametro>
            <parametro>
                <clave>13</clave>
                <nombre>ejemplo_declaracion</nombre>
                <valor>
                    var $nombre = "";   // nombre de la persona
                    var $apellido = "";   // apellido de la persona
                    var $edad = 0;        // edad de la persona
                    var $sexo = "";        // sexo de la persona
                </valor>
            </parametro>
            <parametro>
                <clave>14</clave>
                <nombre>expresion_metodo_pulico</nombre>
                <valor>(function)\s+(?<name>\w+)</valor>
            </parametro>
        </parametros>
</code>
```

Apéndice D.
Interfaz del usuario

SECCIÓN 1. Pantalla principal de la herramienta

La herramienta se compone de una pantalla principal, la cual tiene un menú principal y un área de trabajo, todas las funciones presentan ventanas de dialogo con el usuario en el área de trabajo. Ver Figura 67.

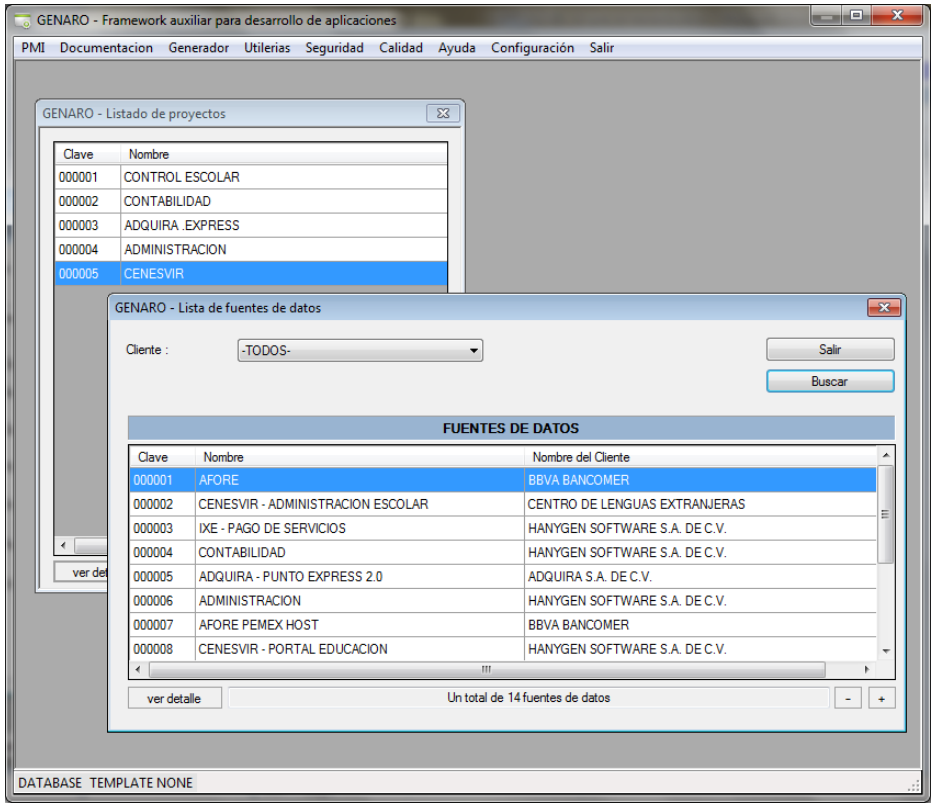

Figura 67. Menú principal y área de trabajo.

SECCIÓN 2. Captura de información de aplicaciones

La información que se captura para una aplicación es lo referente a la base de datos y a las pantallas. Esta captura es la primera que se hace y tiene el objetivo de documentar cada aplicación con los datos generales como: cliente, empresa, comentarios, etc. ver Figura 68.

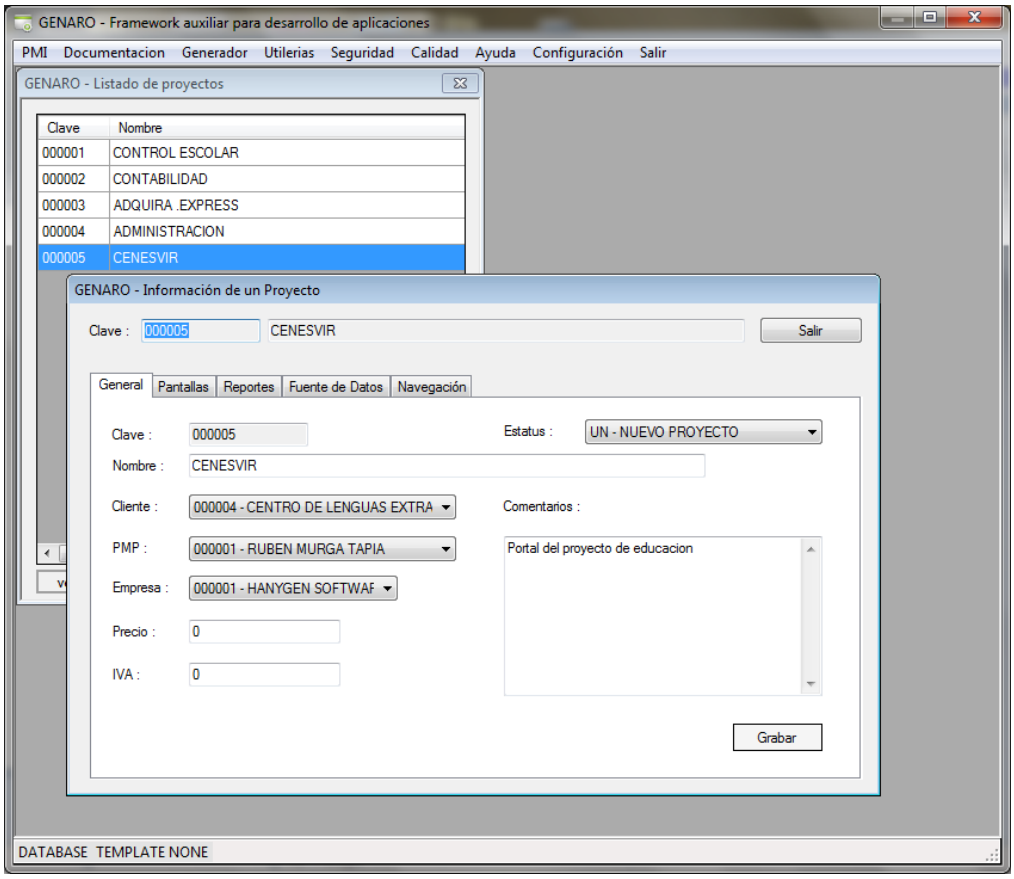

Figura 68. Información de los proyectos.

Una vez que se registra la información general, procedemos a la información a detalle, donde incluimos pantallas, reportes y las fuentes de datos que es el manejo de la persistencia para esa aplicación en particular. Tanto las pantallas como los reportes son especificados hasta el detalle de describir cada campo, botón, validación, evento y navegación de la interfaz. ver Figura 69.

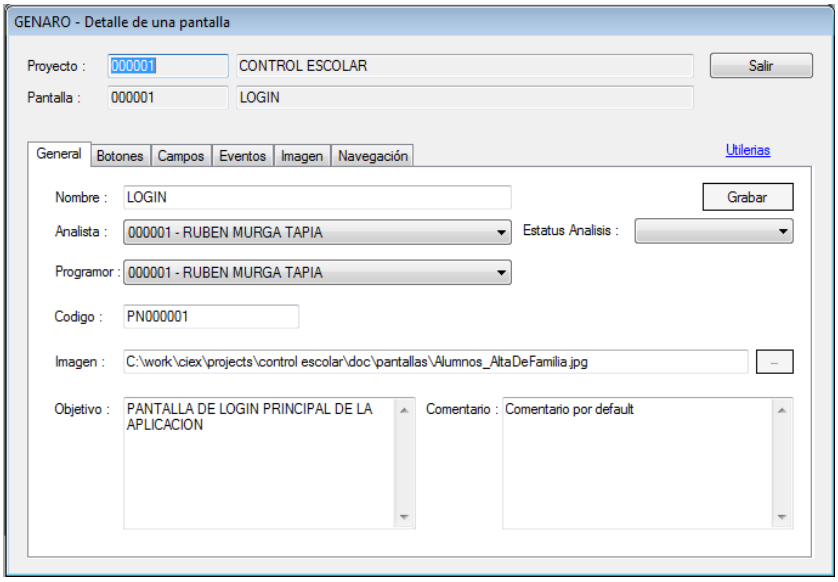

Figura 69. Información de las pantallas.

La captura de información de la base de datos se realiza con la pantalla de captura de la Figura 70:

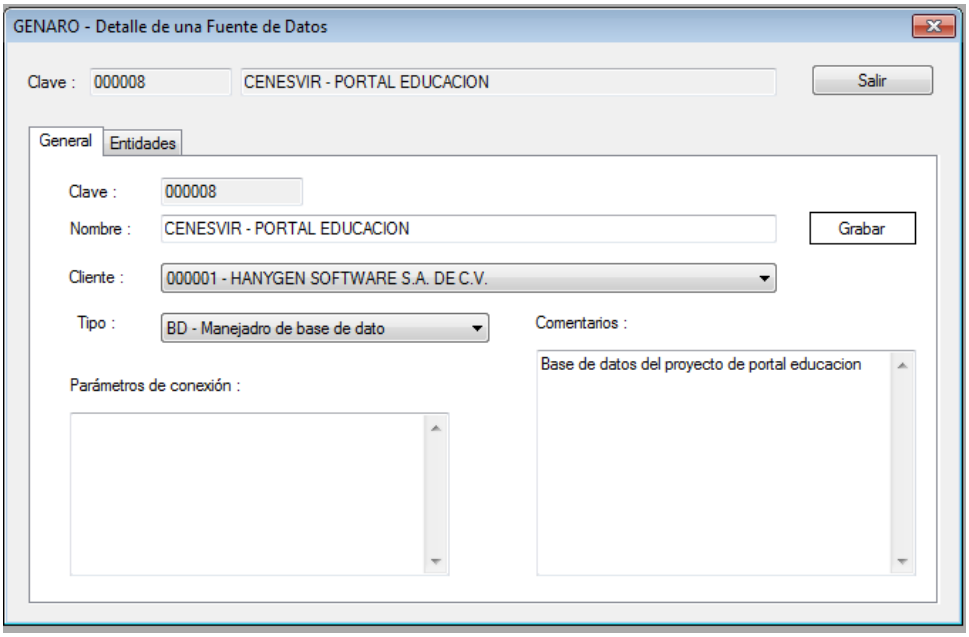

Figura 70. Información de la fuente de datos.

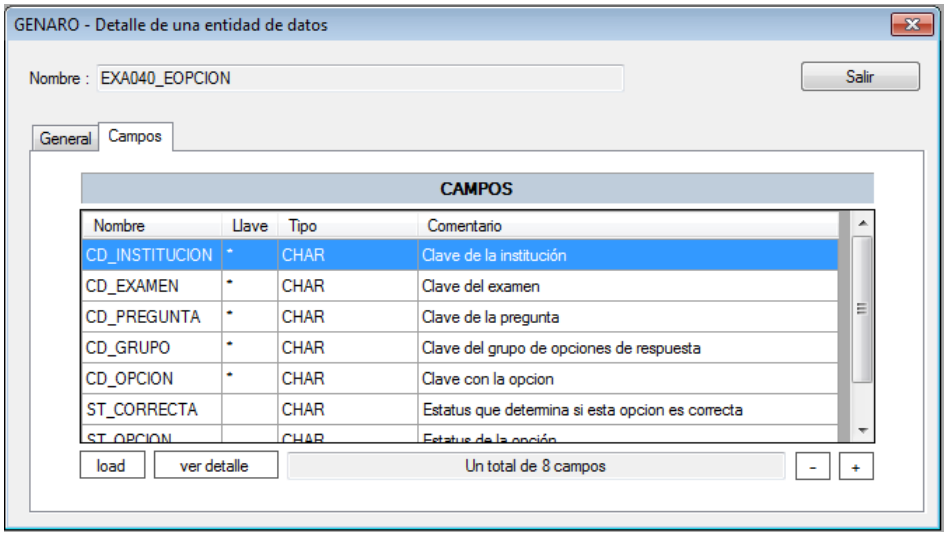

Figura 71. Detalle de cada entidad de datos.

SECCIÓN 3. Proceso de carga automático

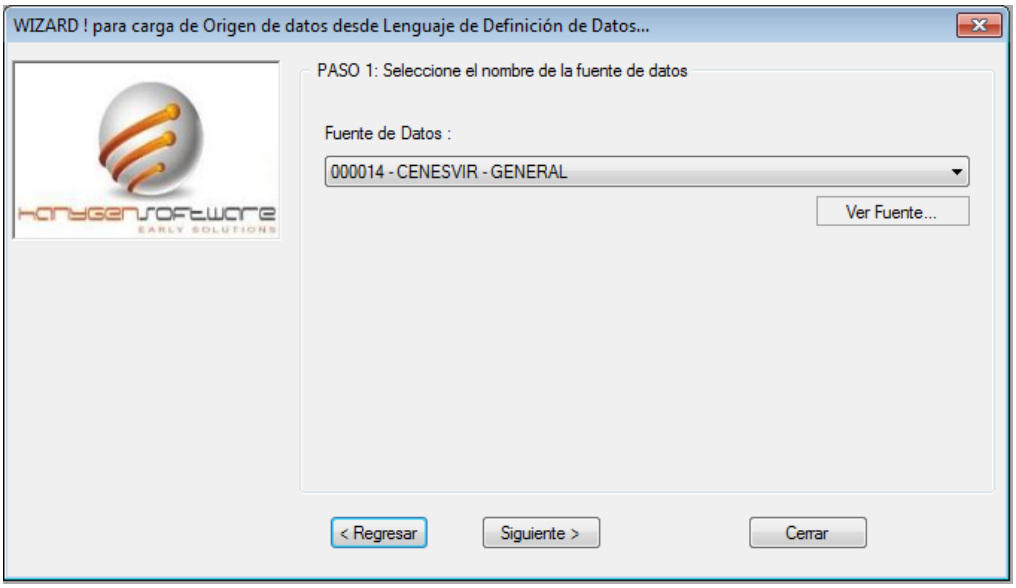

Figura 72. Seleccionando la fuente de datos.

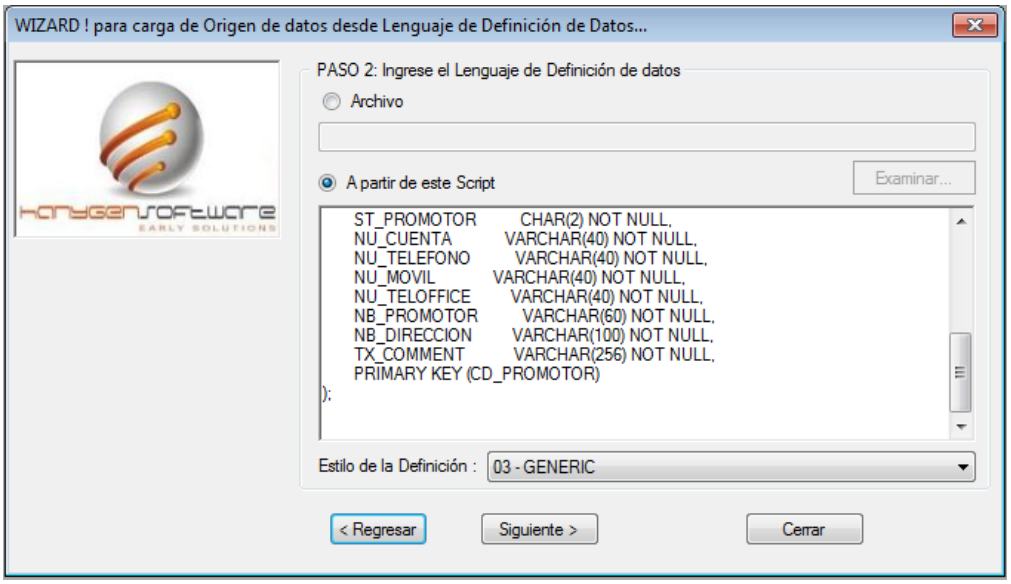

Figura 73. Integrando el script del DDL en archivo o pantalla.

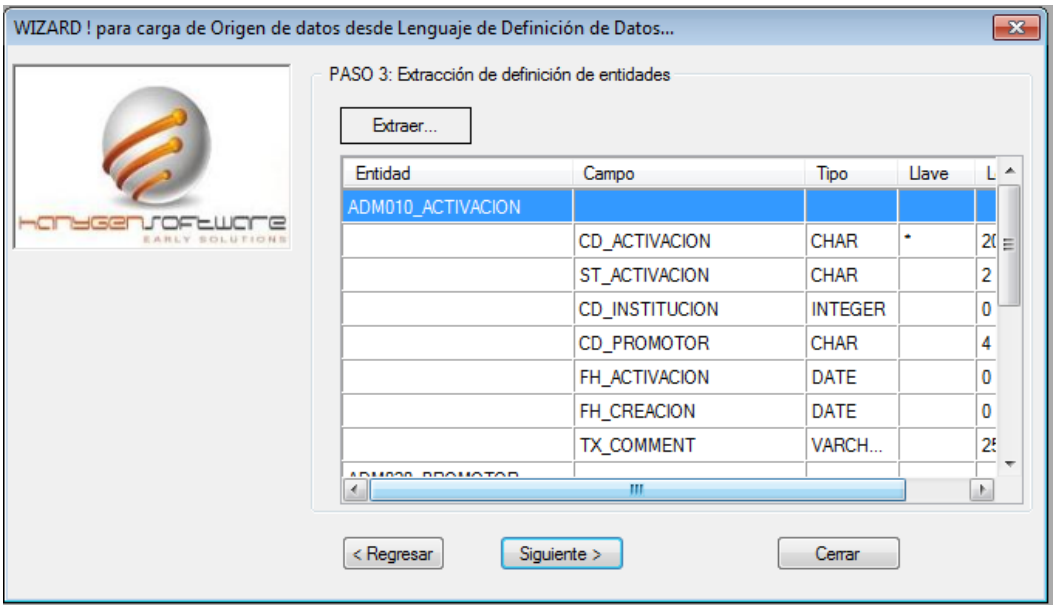

Figura 74. Extrae el detalle de la base de datos.

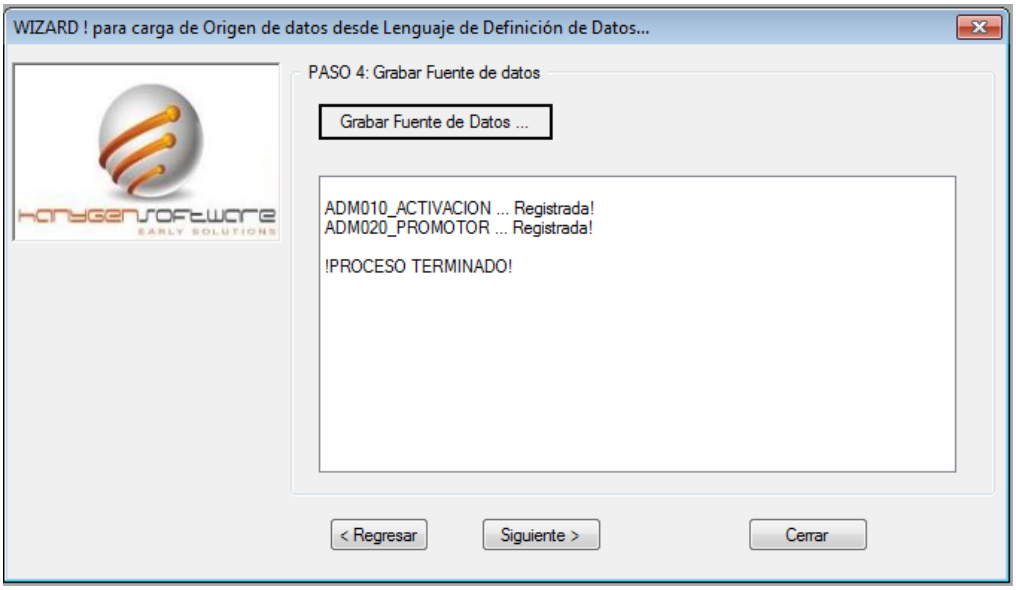

Figura 75. Termina y graba.

SECCIÓN 4. Documentación automática

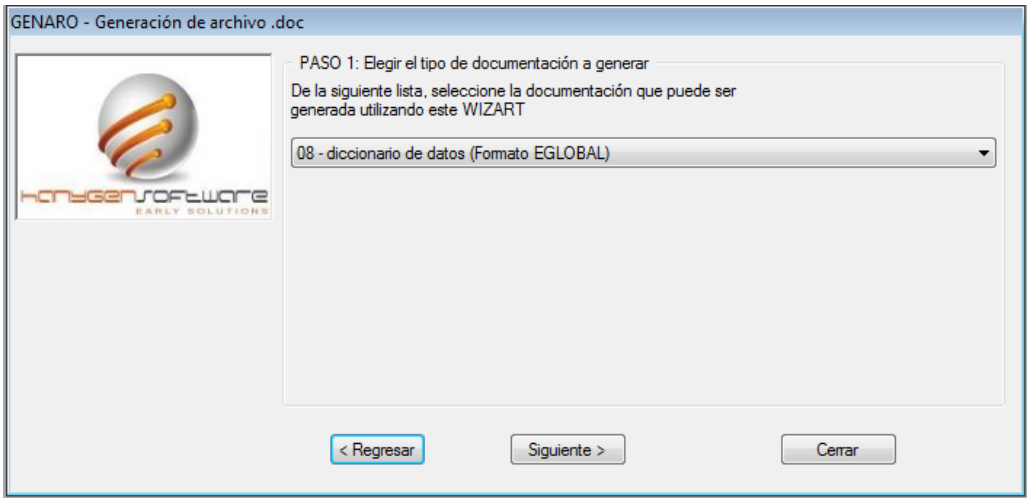

Figura 76. Selecciona el formato de documentación.

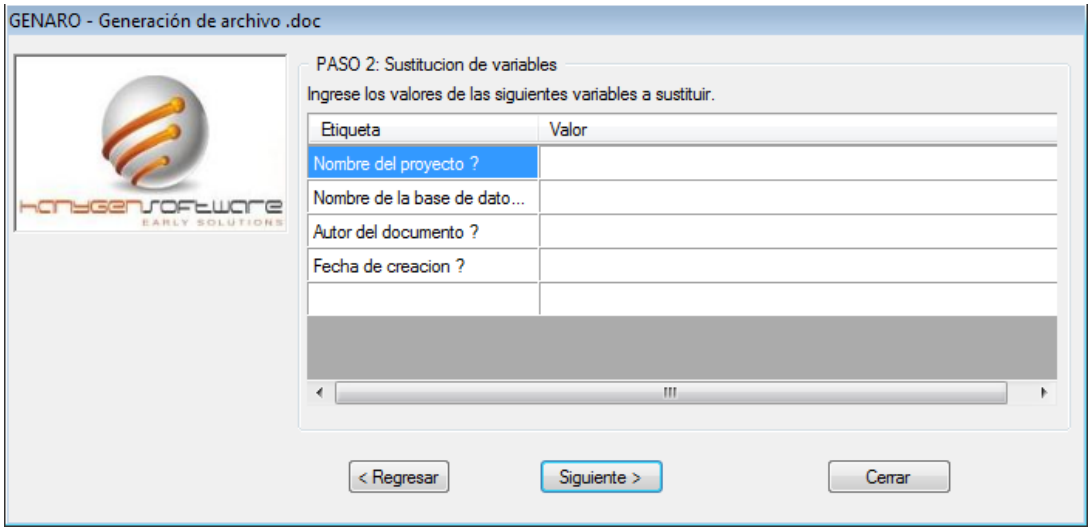

Figura 77. Captura de valores en variables.

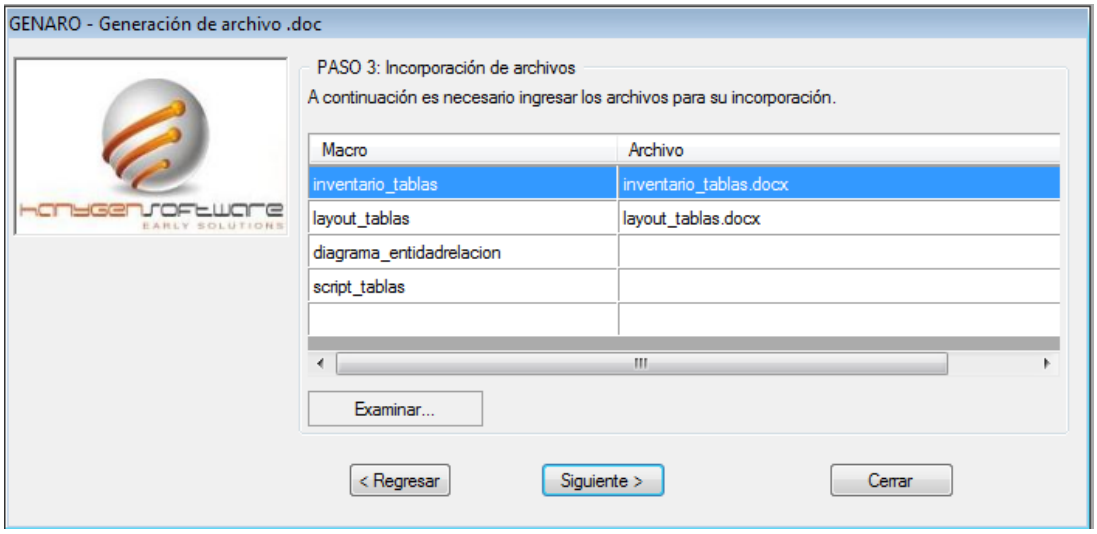

Figura 78. Integrando algunos archivos MACRO.

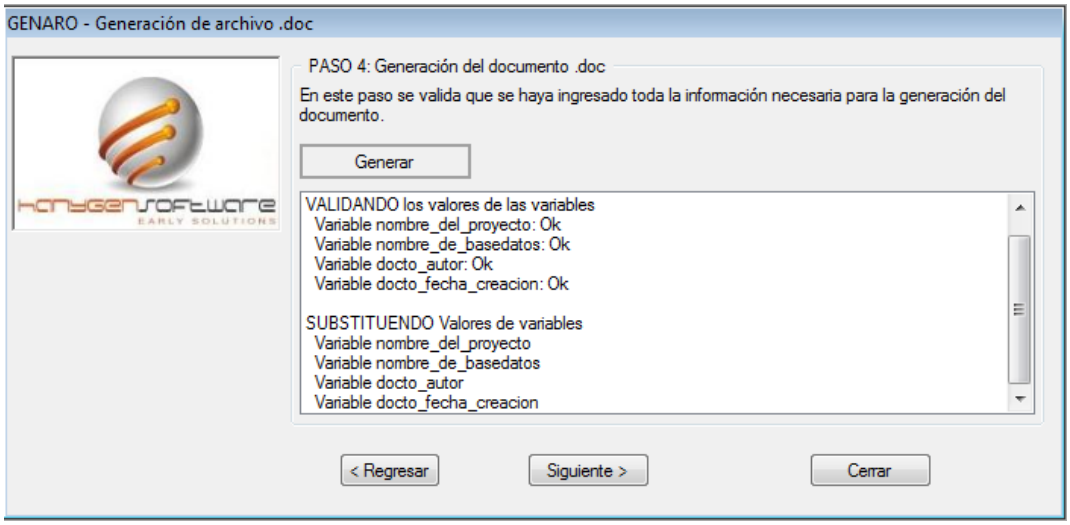

Figura 79. Generación de un documento.

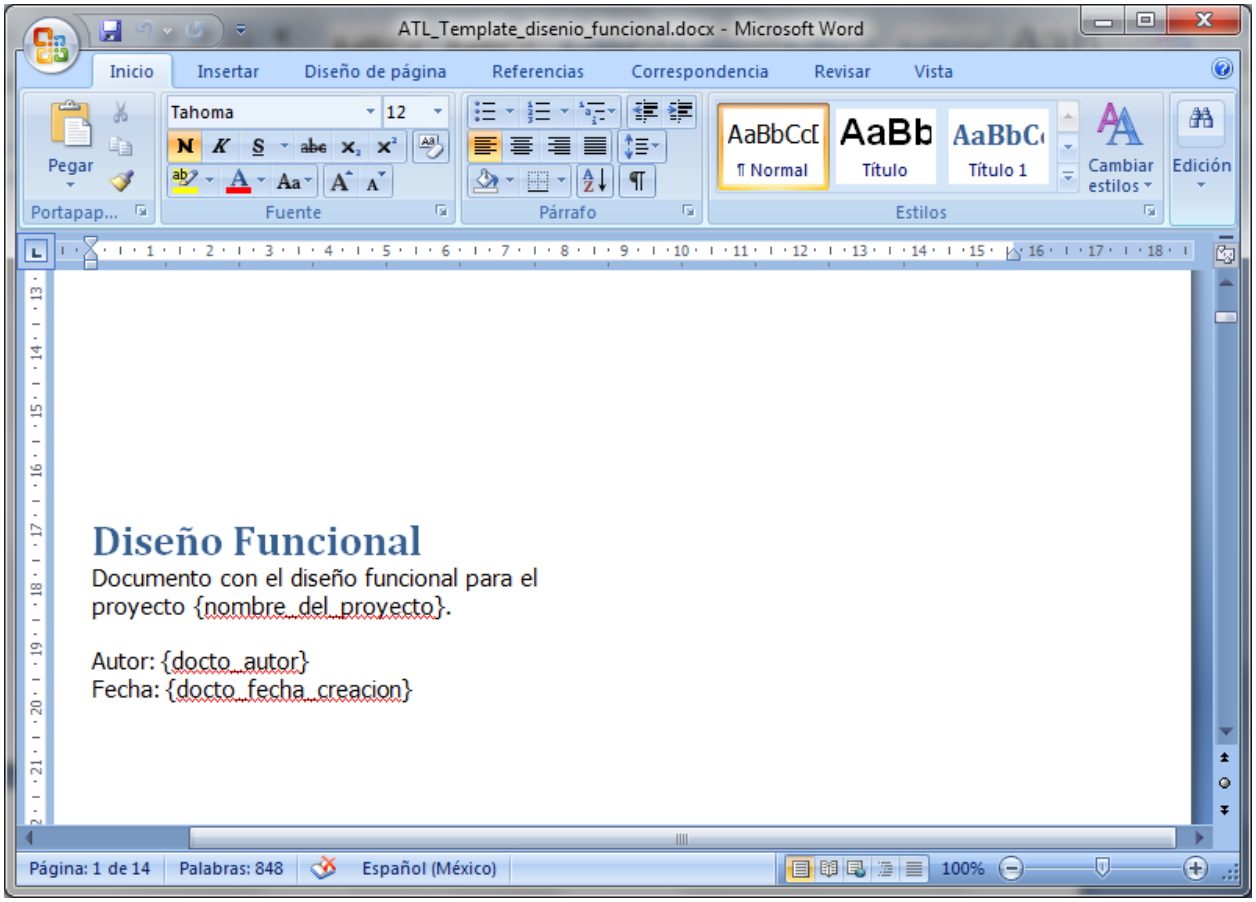

Figura 80. Sustitución de valores en documentos Word.

SECCIÓN 5. Generación de código *Back-End*

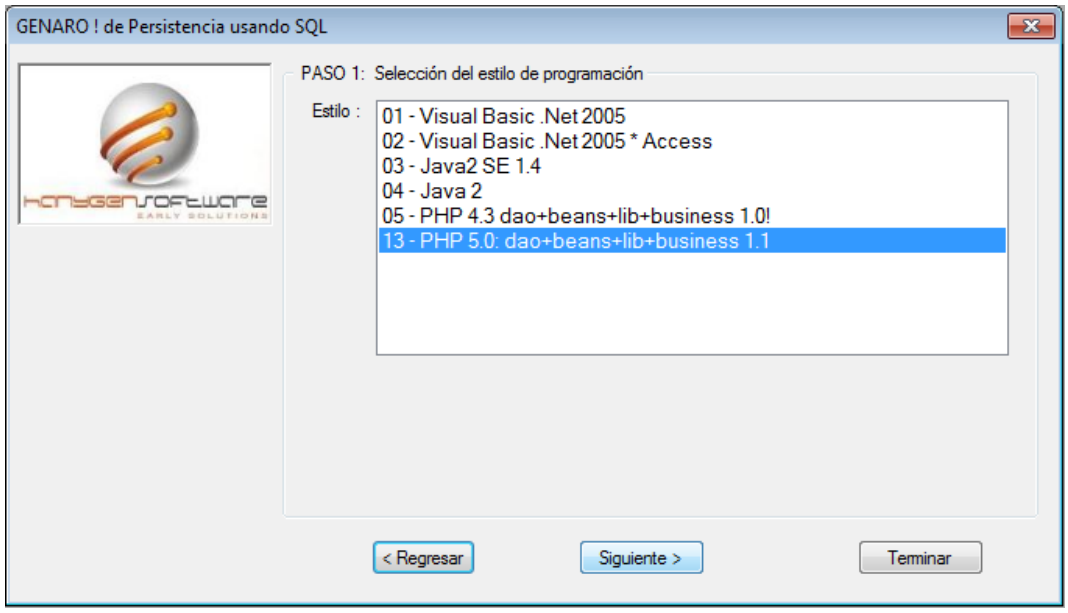

Figura 81. Selección del estilo de programación.

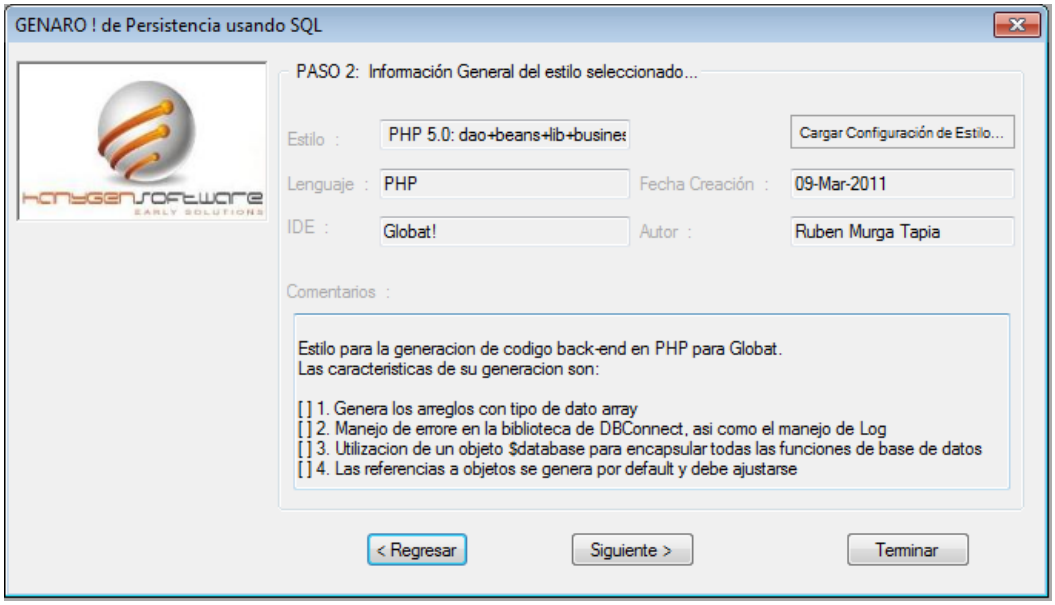

Figura 82. Información del estilo de programación.

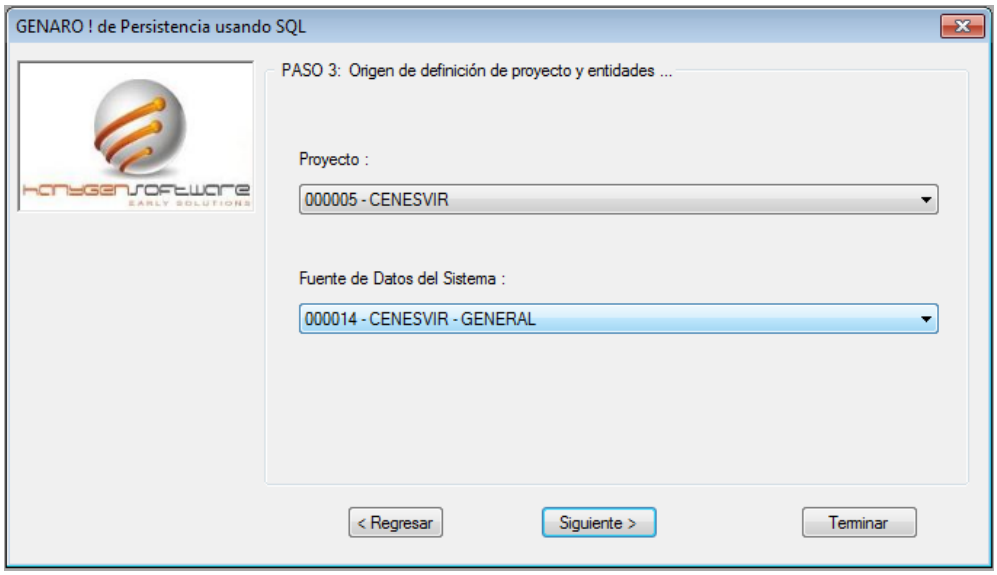

Figura 83. Selección de proyecto y fuente de datos.

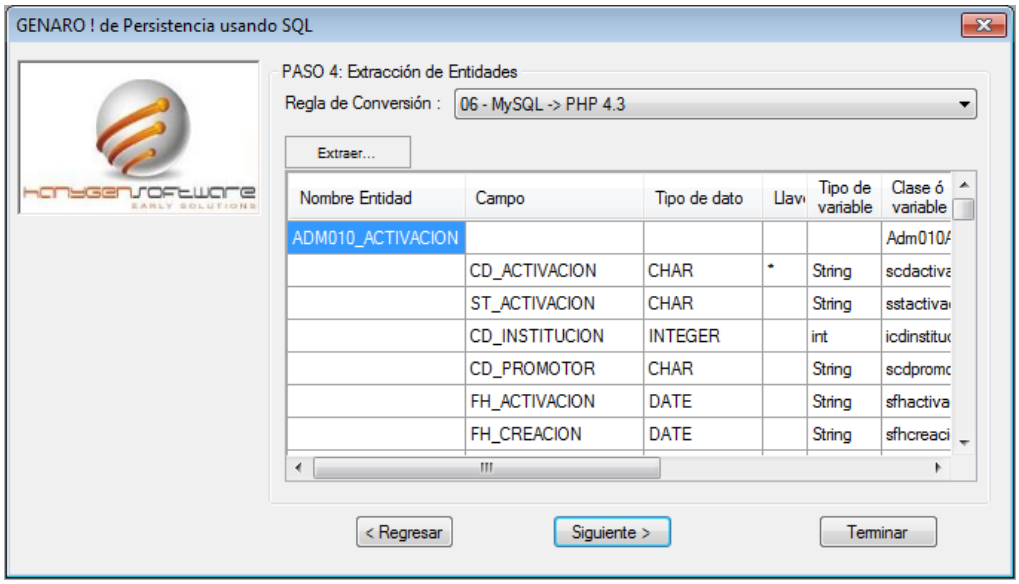

Figura 84. Configuración de las entidades.

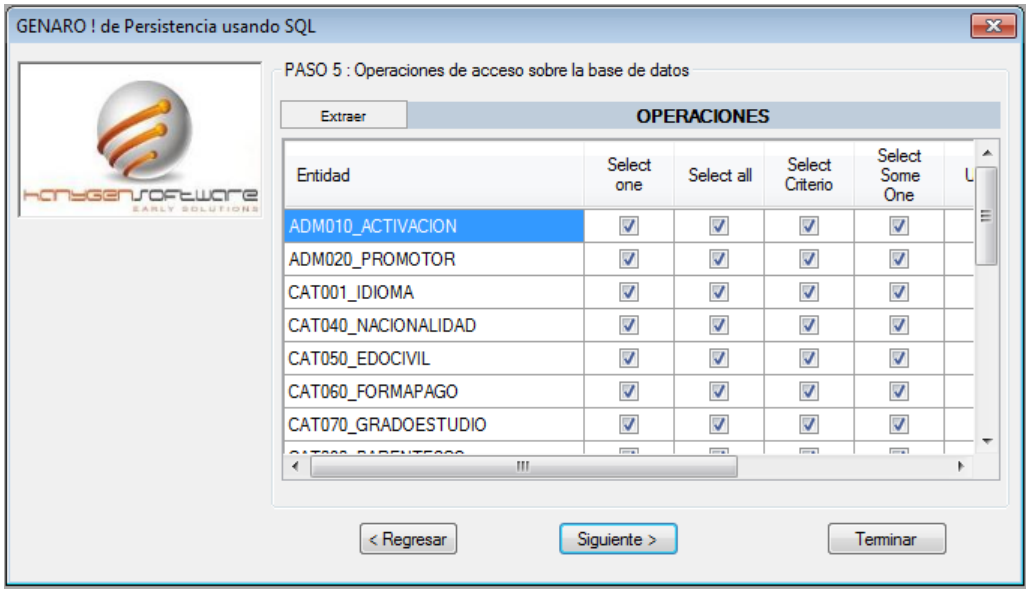

Figura 85. Selección de operaciones de acceso.

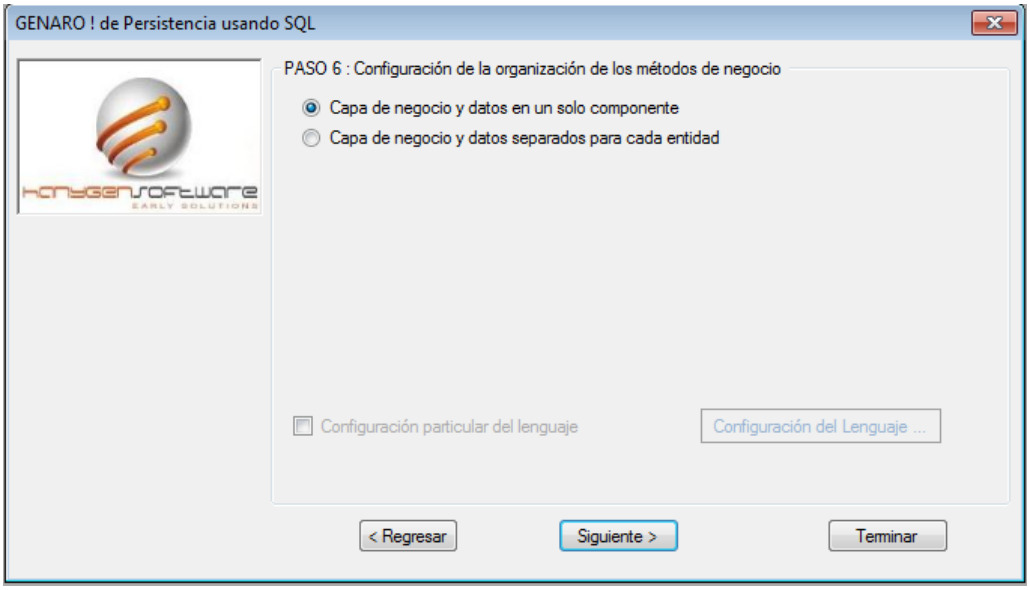

Figura 86. Configuración particular de estilo.

Figura 87. Nombre de componentes y directorio de productos.

SECCIÓN 6. Generación de código *Front-End*

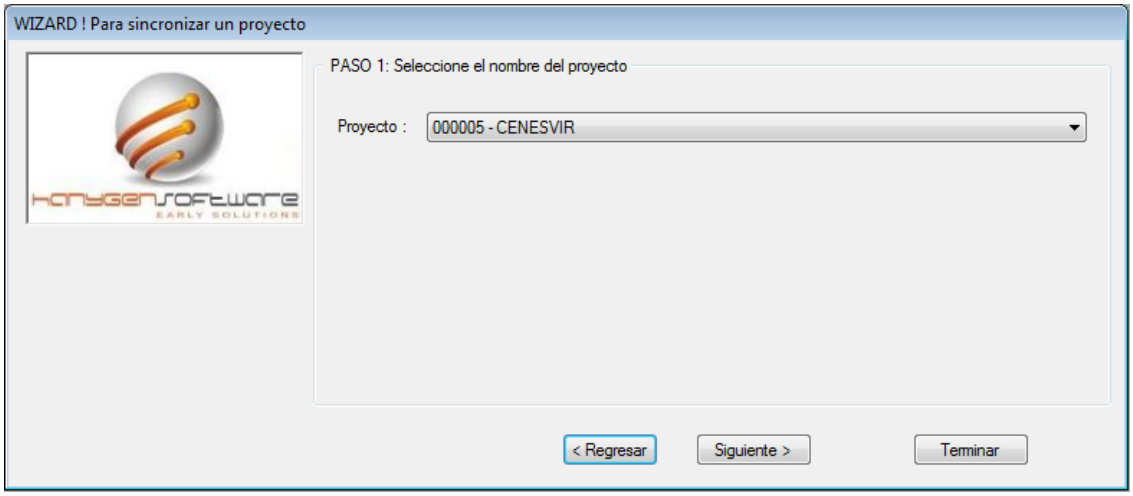

Figura 88. Selección de proyecto.

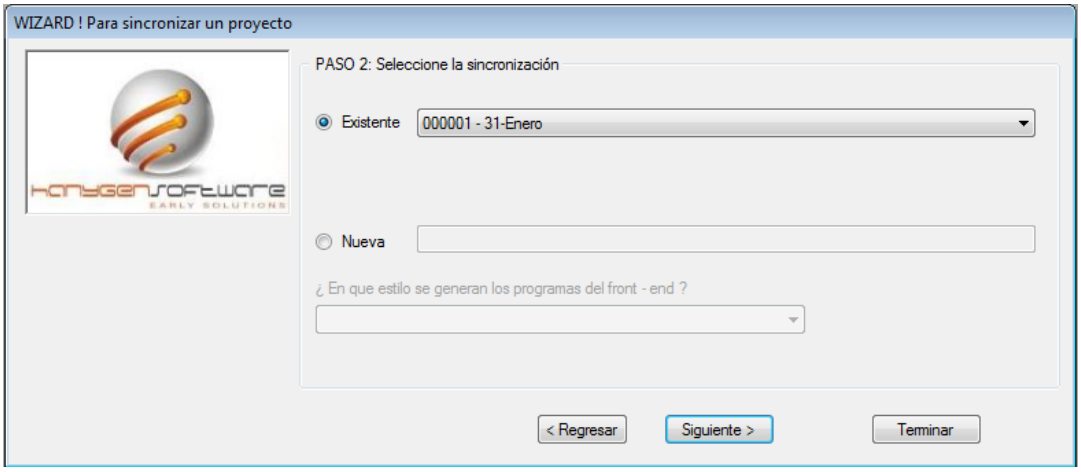

Figura 89. Seleccionando la sincronización.

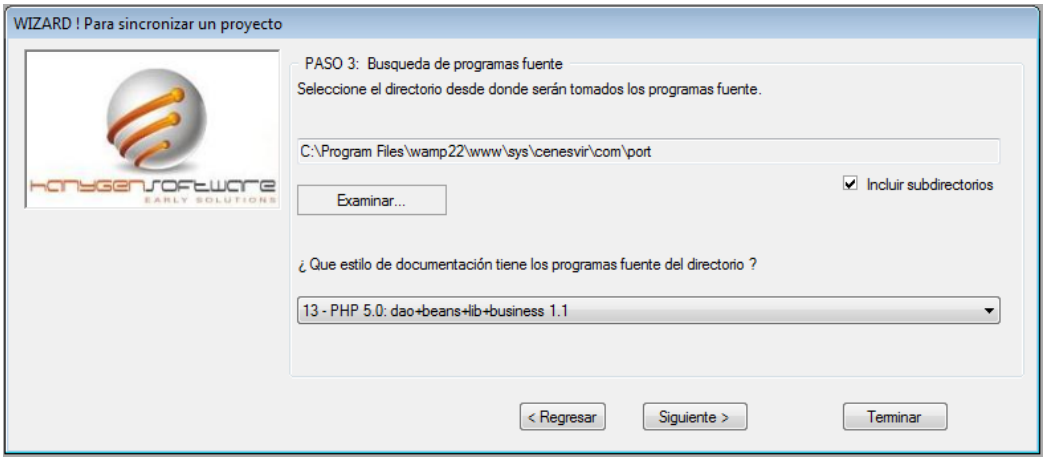

Figura 90. Búsqueda de componentes Back-End.

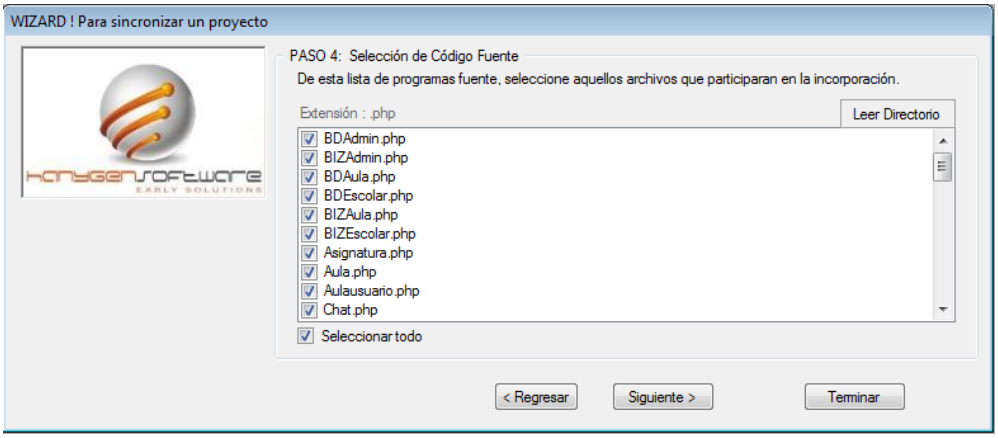

Figura 91. Selección de componentes Back-End.

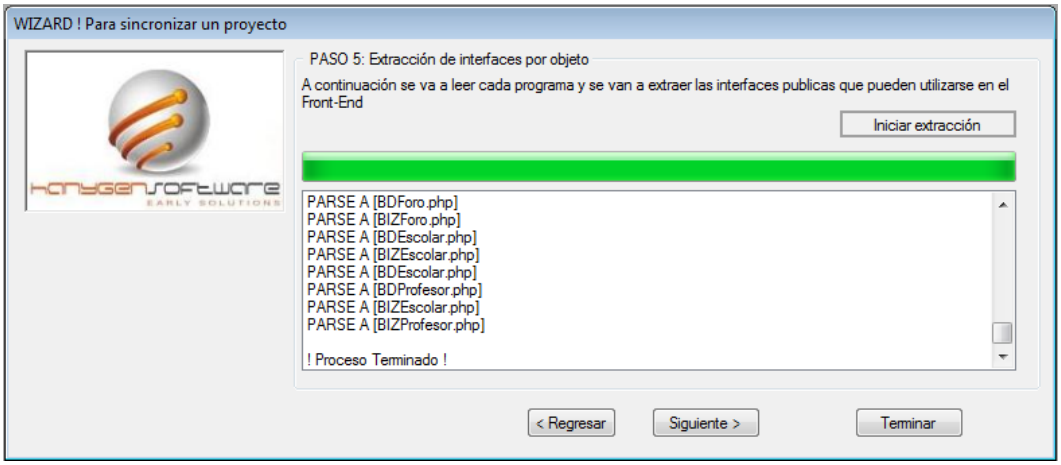

Figura 92. Se extrae la interfaz de cada componente.

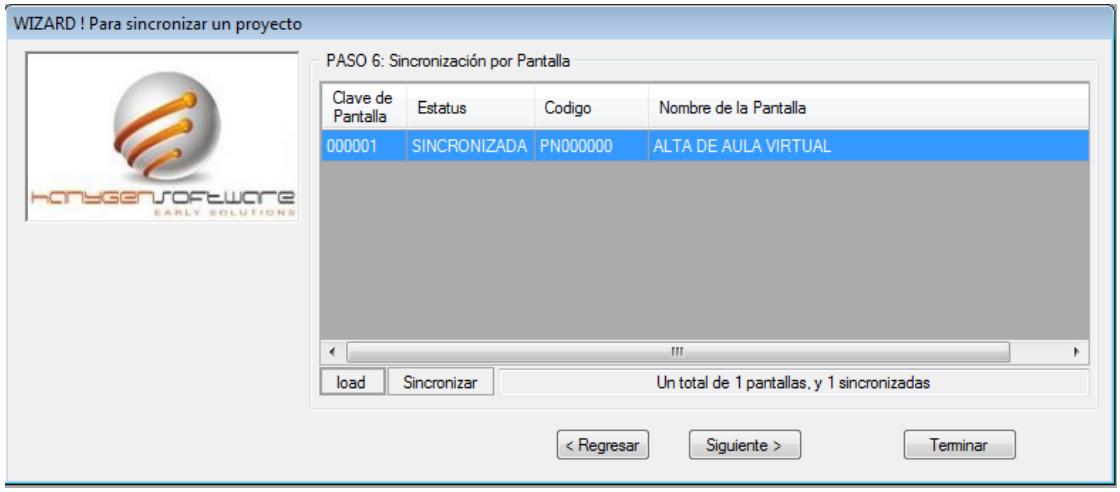

Figura 93. Seleccionando pantalla para sincronizar.

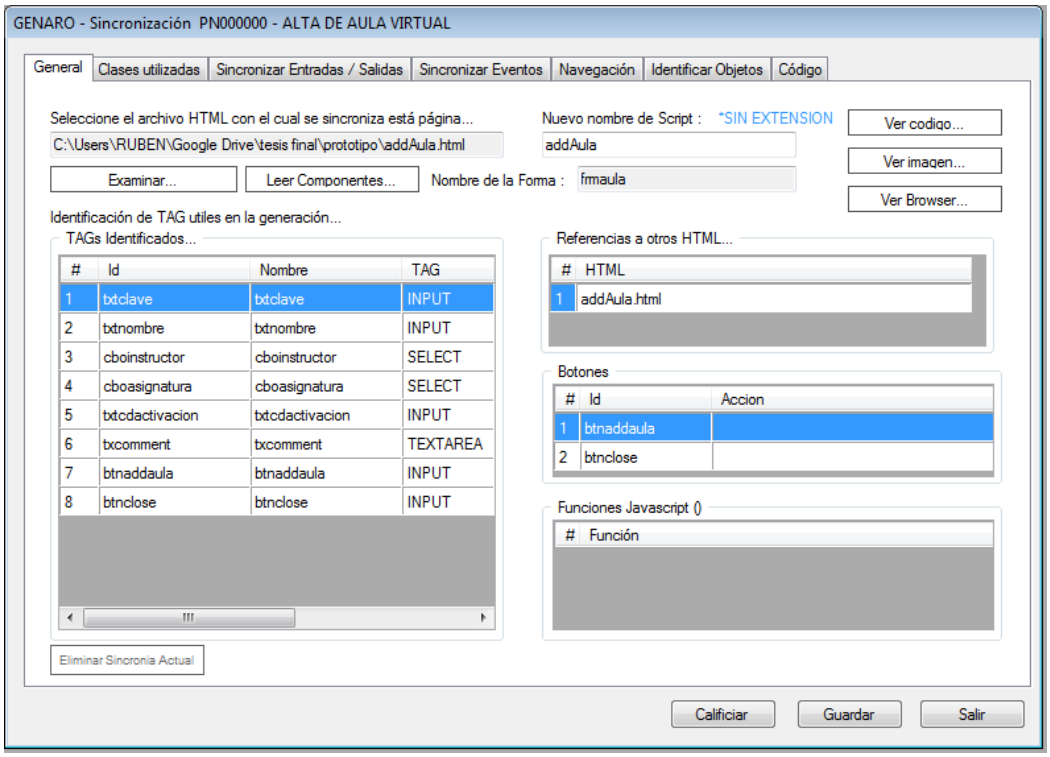

Figura 94. Sincronizando pantalla entre prototipo y clases.

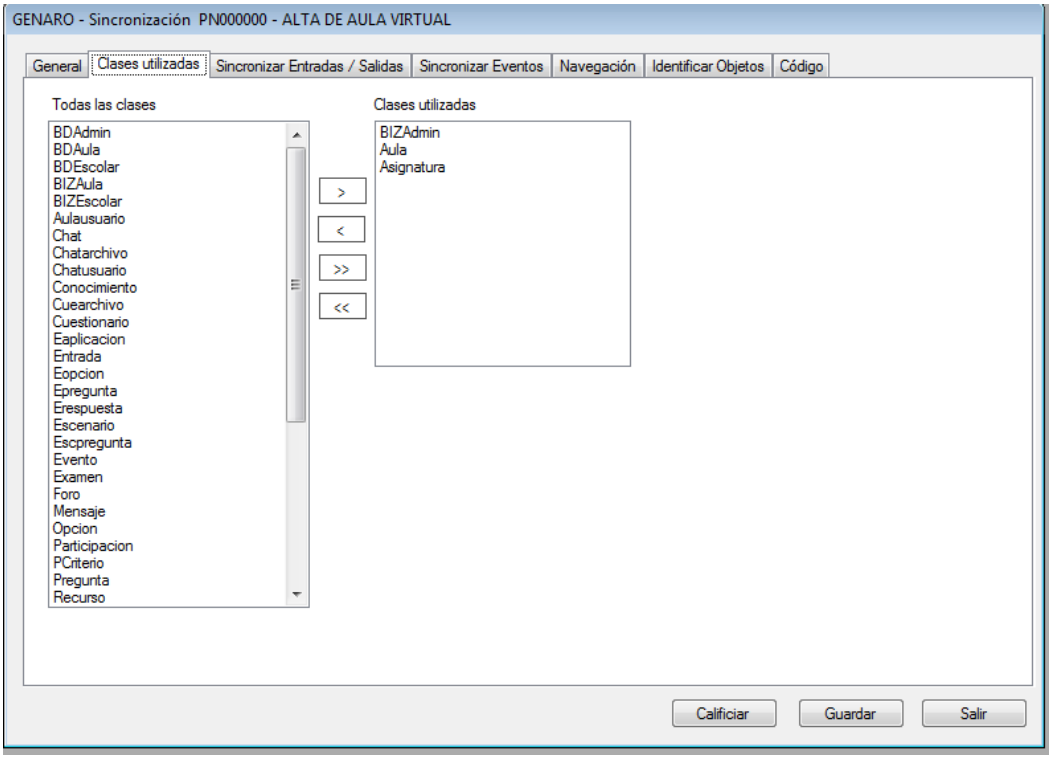

Figura 95. Seleccionando clases participantes.

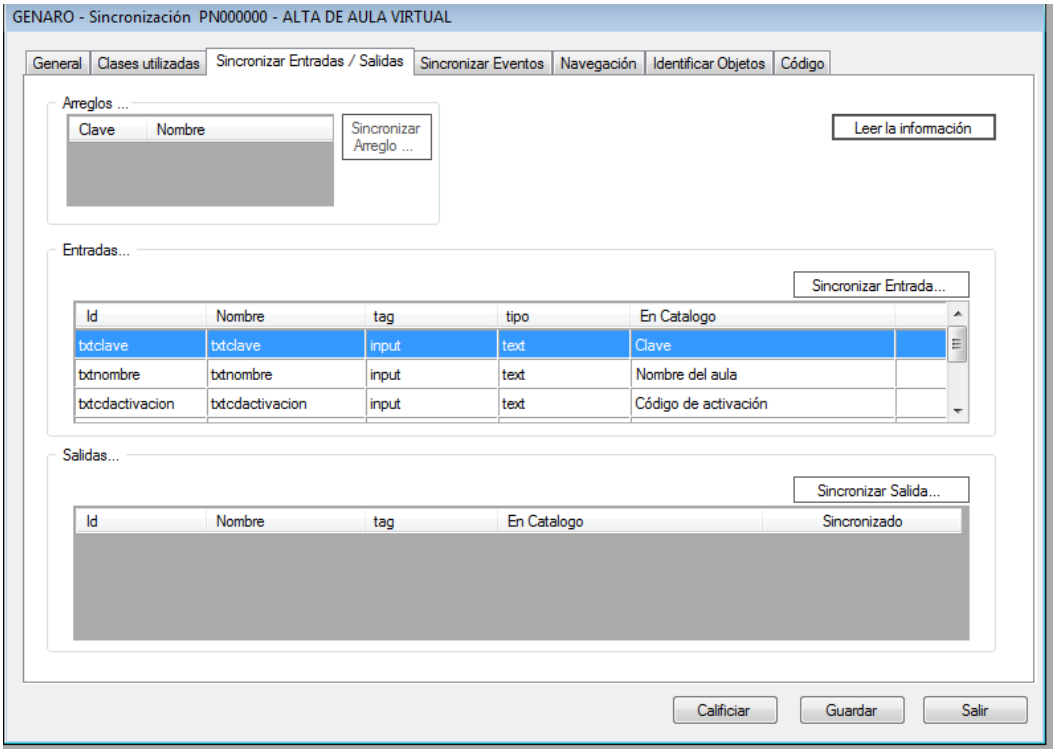

Figura 96. Sincronizando las entradas y salidas.

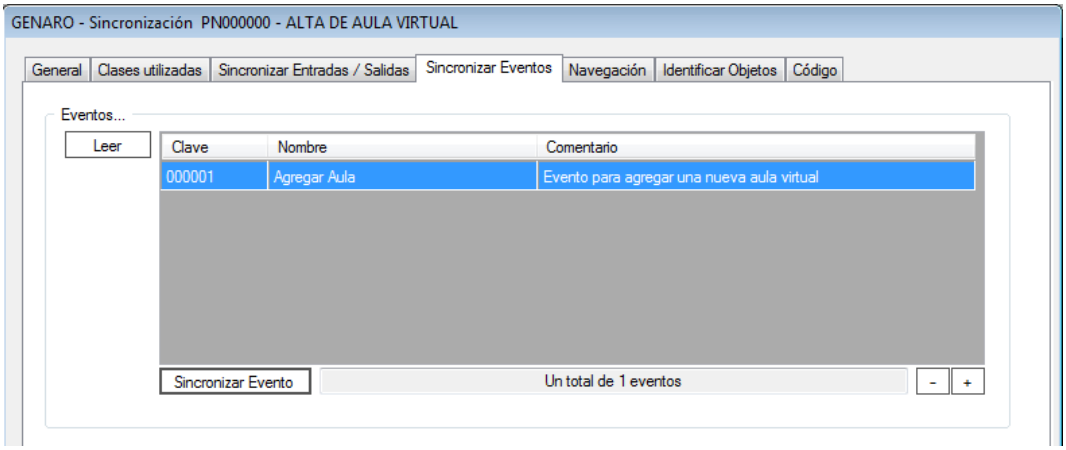

Figura 97. Sincronizando los eventos.

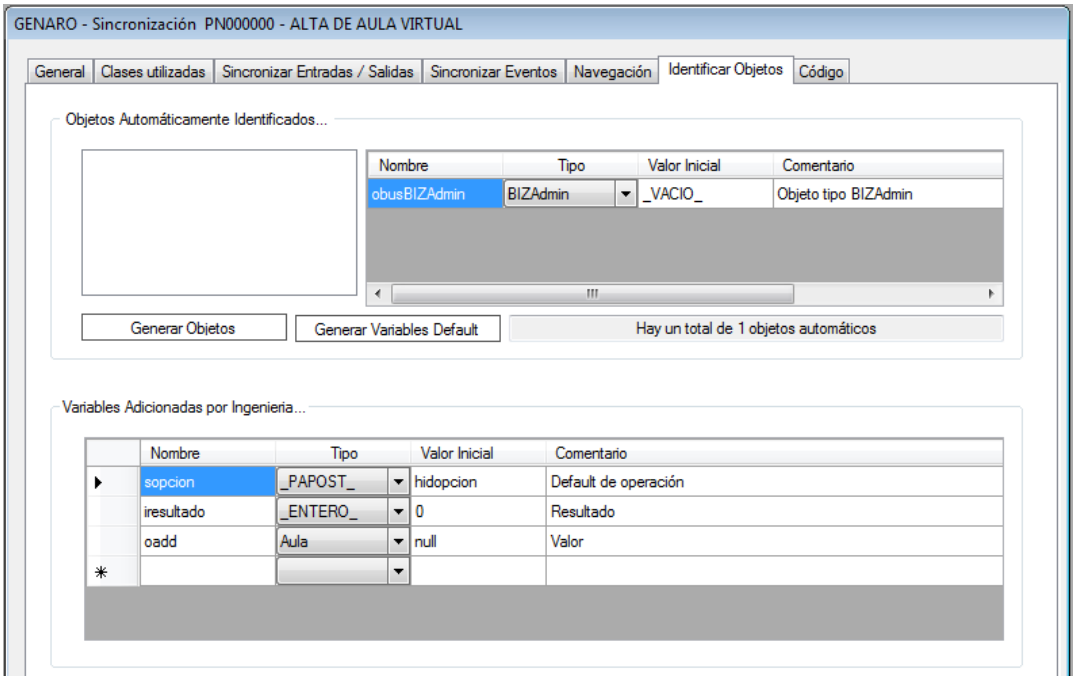

Figura 98. Creación e inicialización de objetos y variables.

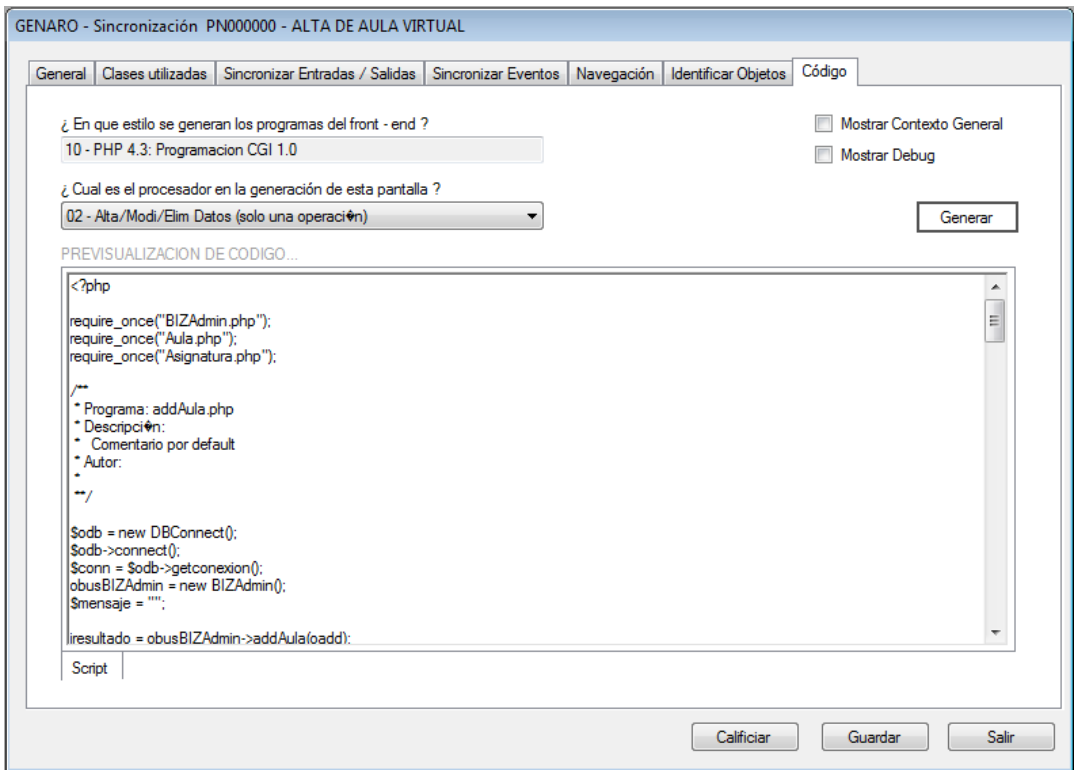

Figura 99. Generación de código.